Politecnico di Milano
V Facoltà di Ingegneria
Master of Science in Computer Engineering
Dipartimento di Elettronica e Informazione

Java Interactive Virtual Environment System (JIVES)

a Java-based Multi-user Modular Networked Virtual Environment framework

Relatore: Prof. Thimoty BARBIERI

Tesi di Laurea di:

Adriano DALPANE	Matr. 736581
Simone SEGALINI	Matr. 739362

Anno Accademico 2010-2011

The computer programmer is a creator of universes
for which he alone is responsible.
Universes of virtually unlimited complexity can be created
in the form of computer programs.

Joseph Weizenbaum

Contents

List of Figures

List of Code Snippets

List of Tables

Abstract

This paper presents the design and implementation of Java Interactive Virtual Environment System (JIVES), a Java-based Multi-user Modular Networked Virtual Environment framework for web and desktop applications. The whole architecture is subdivided into three main layers, ordered as follows: the Core implements the abstract features that are commonly required in the creation of Virtual Environments; the Implementors layer consists in modules that can be contributed as building blocks in order to realize such features; the application layer that defines its own scripting language that allows the programmer to estrange from lower levels implementation details. The current state of development offers a Peer-To-Peer network and a 3D visualization module; along with these we present a demonstrative application script. An innovative HotSpot-based Item Inventory Management System integrates a meaningful way to combine Inventory items and allows JIVES to differentiate from its predecessors. Finally we propose a reflective evaluation of the entire project and we discuss possible future development directions. JIVES aims at being a trade-off solution to a complex problem: completely Open Source, having an experimental design, being extensible, flexible and reusable, it can be considered a good research result in the world of Virtual Environment frameworks.

Estratto

Questa tesi presenta il design e l'implementazione del Java Interactive Virtual Environment System (JIVES) framework, un sistema multi-utente modulare basato su Java che offre funzionalità di rete, attraverso il quale si possono realizzare applicazioni per web e desktop. L'intera architettura è suddivisa in tre livelli principali, ordinati come segue: il Core implementa le caratteristiche astratte comunemente ritenute essenziali nella creazione di ambienti virtuali; il livello degli Implementors consiste in moduli che contribuiscono alla realizzazione di tali caratteristiche; il livello applicativo definisce il proprio linguaggio di script in maniera tale che il programmatore rimanga completamente estraneo ai dettagli implementativi dei livelli inferiori. Allo stato attuale, JIVES offre un modulo di rete basato su architettura Peer-To-Peer ed un motore di visualizzazione 3D; assieme ad essi viene fornito lo script di un'applicazione dimostrativa. Un sistema innovativo di gestione dell'inventario basato su HotSpots integra una modalità significativa per combinare oggetti dell'inventario e permette a JIVES di differenziarsi dai sistemi finora sviluppati. Infine proponiamo una valutazione dell'intero progetto e discutiamo le possibili direzioni di sviluppo futuro. JIVES punta ad essere una soluzione di trade-off ad un problema complesso: completamente Open Source, estensibile, flessibile, riutilizzabile e caratterizzato da un design sperimentale, può essere considerato un buon risultato di ricerca nel panorama dei Virtual Environment frameworks.

Chapter 1

Introduction

An exact definition of a Virtual Environment (VE) is difficult to find. A common definition denotes a Virtual Environment as a computer-based simulated environment intended for its users to inhabit and interact via avatars. This habitation usually is represented in the form of two or three-dimensional graphical representations of humanoids (or other graphical or text-based avatars). Blaskovich [5] defines a Virtual Environment as synthetic sensory information that leads to perceptions of environments and their contents as if they were not synthetic.

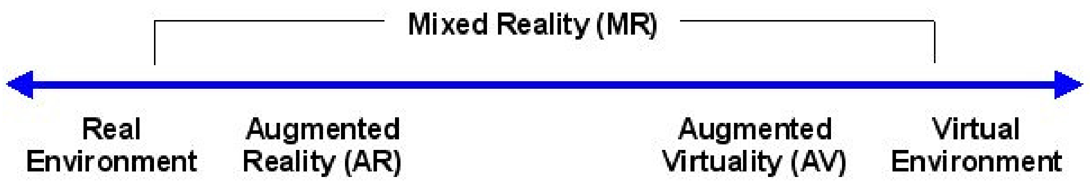


Figure 1.1: Reality-Virtuality Continuum

According to the definition of the Reality-Virtuality continuum, shown in Figure 1.1, in which the range scales between the completely virtual, a Virtuality, and the completely real, a Reality, Virtual Environments lie on the right side of the continuum. Between the two extremes of the continuum, the pure Real Environment and the pure Virtual Environment, there exist numerous variations, both theoretical and applicative, which mix to various degrees the elements of both the Real and Virtual Environments: the spectrum in which this takes place is commonly referred to as Mixed Reality. Mixed Reality applications can be based on real elements, with the addition of a number of synthetic ones generated by the computer, in which case the talk is of Augmented Reality. Alternatively the application can be almost entirely based on a synthetic environment, with a number of real elements supporting it, in which case it is referred to as Augmented Virtuality.

Virtual Environments applications can be extremely different from each other leading to a categorization into four main classes:

- **Collaborative Virtual Environments (CVEs)** describe Virtual Environments that involve more than one user, with avatars interacting with each other. With high bandwidth and Internet access, Virtual Environments that allow for greater multi-user interactivity have become widely available in recent years.

- **Immersive Virtual Environments (IVEs)** increase the user's sense of presence actually being within it. IVEs typically require special equipment such as a head mounted display or a project equipment situated in a room. IVEs track a user's head and body position, facial expressions and gestures, and other information, thereby providing a much information about the zone or the item of the Environment in which the user is focusing the attention.

- **Massively Multiplayer Online Role-Playing Games (MMORPGs)** describe multi-player games which are capable of supporting thousands of players simultaneously. They are often based on fantasy themes [6, 41, 62].

- **Multi-user domains (MUDs)** are primarily text-based environments that were the predecessors of modern graphical Virtual Environments.

In particular, Collaborative Virtual Environments include a category of systems in which individuals cooperate in a network according to the paradigms of released reality, placing these applications into a category referred to as desktop virtuality (or semi-immersive virtuality), in which the only two items that are necessary are a good graphics card and a good personal computer. The advent of standards such as VRML for 3D representation on the Web have allowed to also develop other types of applications that are entirely Web-based. This category is often termed Net-VE (Networked Virtual Environments), and also includes a subgroup called Web-VE (Web-Based Virtual Environments), which consists solely of Environments that use Web technologies instead of own technologies. The salient characteristic of these systems is that they are network-based and therefore are essentially concerned with establishing strong collaboration between agents that share an Environment. According to the MRIC model [66] modern Net-VEs display little if any integration with the real environment, while their degree of Released Reality and Immersivity differs according to the type of application.

1.1 Development of a Virtual Environment system

A Virtual Environment presents a challenging problem with regard to the development of an underlying system. The problem domain presents itself being vast, requiring diverse areas of expertise, which may range from networks to psychology. This complexity makes the development of a Virtual Environment system a difficult task to achieve with a high cost in resources. Oliveira [46] asserts that the wide applicability of a Virtual Environment, such as scientific visualization, socializing, training, gaming, produce a set of requirements that make it very difficult to build a single system to fit all needs. Traditionally, the result has been the creation of monolithic systems that are highly optimized to a particular application, without possibility of re-usability with a different purpose. In the last decade, the main trend in the Virtual Environment community has been for a new Virtual Environment system to be developed every time it was necessary to use one. The trend in the games development industry is even worse, where the production cycles of each application traditionally involve game design, technology development and content creation. In some cases the particular requirements of each game usually involves significant changes in the code, leading to total re-development of the game engine. The main problems mentioned by Oliveira [46] which have affected the development of Virtual Environment systems during the last decade are here reported:

- **Non-Extensibility.** The design of most Virtual Environment systems is tightly coupled with the initial requirements, thus resulting in monolithic architectures where any changes or modifications are infeasible. The architectures of more recent versions of some systems do present a modular design, but continue to make it difficult to extend the core functionality if it was not foreseen in the original design. Thus, any changes require significant, if not total, re-engineering of the underlying system. In most cases, the most cost/effective solution is the creation of a new system.

- **Steep Learning Curve.** The complexity of a Virtual Environment system, with its tightly coupled nature, makes it difficult for a developer to haul any benefits without becoming an expert. Unfortunately, the learning curve associated to a system is traditionally exponential. This results in a selected few being sufficiently proficient with a particular system, normally the creators and maintainers. Recently, the adoption of different scripting languages has improved Virtual Environment development making it more efficient and accessible.

- **One Stop Shop.** The complexity of VE involves the operation of several different sub-systems, such as rendering, networking, database. Although modularity may influence the design of each subsystem, their operation remains tightly coupled to each other. Consequently, the result is a monolithic architecture, albeit modular.

- **"Not Invented Here" and "Reinventing The Wheel".** These syndromes imply the expenditure of resources on the re-emergence of existing technology in building a Virtual Environment system. Consequently, exploring new approaches becomes quite limited.
- **Poor Scalability.** Most Virtual Environment systems aimed at collaboration claim to support in theory thousands of users, when in reality all documented experiments in Collaborative Virtual Environments (CVEs) do not go beyond a few dozens. With the online game community, the number of user base is reported to be larger at the expense of significant large budgets to increase the network and computational resources. This approach is less than ideal since notoriously the Client/Server architectures do not scale.

Only recently the development of Virtual Environment systems is considering these issues, adopting different solutions such as modularity, peer-to-peer architecture, scripting languages, code re-usability. Chapter 2 illustrates the main technologies adopted in the development of a Virtual Environment system, while Chapter 3 presents a brief analysis of the previous work in the literature, along with the introduction of our new Virtual Environment system, JIVES. Chapter 4 includes the motivations and the targets behind the choice of creating the JIVES middleware, while Chapter 5 is subdivided into two main sections, the first dedicated to the design of JIVES, while the latter to the implementation of the framework. The implementation section is in turn divided into nine parts, each of them involving a specific piece of JIVES: the Core, the Persistence Layer, the Actions and Events Management System, the Dialogues Manager, the Bag Graph, the Networking Layer, related to the JXSE [54] technology, the Middleware Layer, which includes the description of the Shell Implementor and the jME [31] Implementor, and finally the Application Layer, which describes the JIVES scripting language, JiveScript. Chapter 6 regards Development and Usage: the

Development section offers a Developer's Manual; while the Usage section includes a User's Manual and the accurate description of how the JiveScript Demos have been developed and structured. Chapter 7 illustrates the Performance Metrics, both qualitative and quantitative, and the Evaluation based on the key points of the Design phase and the Performance Metrics. Chapter 8 draws the conclusions and possible future works. The UML class diagram, the JiveScript Demos written to test the proper functioning of the JIVES Virtual Environment system and the Rendezvous directory server page are fully reported in the Appendices.

Chapter 2

Technological background

There are several technologies that allow the construction of distributed virtual reality systems. In the next subsections will be considered only some of these technologies, with particular attention to those most used in the last years. It is important to remark that all the technologies presented in this chapter were taken into account for the JIVES project, but only some of them were actually used.

2.1 VRML

The Virtual Reality Modelling Language (VRML) [70] established a standard for the description of Virtual 3D Environments that can be viewed locally or transferred through the Internet and provides powerful resources for modelling complex 3D scenes. It is responsible for showing the virtual scene to the user. VRML is often used in combination with the Java [51] development language and the bridge used to communicate this language with VRML is the External Authoring Interface [69]. In the last ten years many 3D middlewares, frameworks and virtual worlds were created by using the VRML technology [34, 2, 36, 3].

2.2 X3D

The Extensible 3D (X3D) [71] is the successor to the VRML. It improves upon the latter with new features, advanced APIs, additional data encoding formats and a component-based architecture. X3D has been chosen as the description language of the component geometry due to its flexible XML-encoding, modularization and backward compatibility with VRML [70] data [14, 35]. Furthermore, X3D includes a rich set of primitives for modelling 3D geometry, behaviours and interactions [3].

2.3 Java3D

Java3D [48] is a scene graph-based 3D application programming interface (API) for the Java [51] platform. It runs atop either OpenGL or Direct3D. Java 3D is not only a wrapper around these graphics APIs, but an interface that encapsulates the graphics programming using a true object-oriented approach. A scene is constructed using a tree-structured scene graph that is a representation of the objects that have to be shown. Several projects employed Java 3D [2, 36, 3, 65].

2.4 JOGL

Java OpenGL (JOGL) [53] is a wrapper library that allows OpenGL to be used in the Java [51] programming language. JOGL allows access to most features available to programs developed using C, with the notable exception of window-system related calls in OpenGL Utility Toolkit (GLUT), since Java [51] has its own windowing systems, Abstract Window Toolkit (AWT) and Swing. Over the last years, different graphical engines have been implemented on top of JOGL [33].

2.5 LWJGL

The Lightweight Java Game Library (LWJGL) [55] is an Open Source Java [51] software library for game developers. LWJGL exposes high-performance cross-platform libraries commonly used in developing software applications, such as

OpenGL, OpenAL, OpenCL. The main goal of the LWJGL [55] project is to provide a technology which allows Java [51] programmers to get access to resources that are otherwise unavailable or poorly implemented on the current Java [51] platform. The jMonkeyEngine [31] uses LWJGL as its rendering system.

2.6 OpenGL

The Open Graphics Library (OpenGL) [32] is a standard specification defining a cross-language, cross-platform API for writing applications that produce 2D and 3D computer graphics. The interface consists of over 250 different function calls which can be used to draw complex 3D scenes from simple primitives. OpenGL is a low-level, procedural API, requiring the programmer to specify the exact steps required to render a scene. This contrasts with descriptive APIs, such as Java3D [48], where a programmer only needs to describe a scene and can let the library manage the details of rendering it.

2.7 Direct3D

Direct3D [38] is part of Microsoft's DirectX application programming interface (API). Direct3D is available for only Microsoft Windows operating systems (Windows 95 and above) and Open Source software Wine [11]. It is used to render 3D graphics in performance-based applications, such as games. Direct3D uses hardware acceleration if available on the graphics card, allowing for hardware acceleration of the whole rendering pipeline.

2.8 jMonkeyEngine

jMonkeyEngine (jME) [31] is a high performance Java-based 3D graphics library made especially for modern 3D development, since it uses shader technology extensively. jMonkeyEngine [31] is written purely in Java [51] and uses LWJGL [55] as its default renderer for OpenGL [32] access. OpenGL 2 through OpenGL 4 is fully supported. It provides a 3D scene-graph based API with the latest state-of-the-art features. jME [31] is completely Open Source under the BSD license. The last graphics engines of the Massively Multiplayer Game Research Framework (Mammoth) [33] and the Open Wonderland framework [47] are based on the jMonkeyEngine [31].

2.9 Nifty GUI

Nifty GUI [43] is a Java [51] Library that supports the building of interactive user interfaces for games or similar applications. It utilizes LWJGL [55] for OpenGL [32] rendering. The base GUI layout is defined in XML, and controlled dynamically from the Java [51] code. Nifty GUI helps the developer to layout stuff, display it in a cool way and interact with it. Nifty GUI [43] (the *de.lessvoid.nifty* package) is well integrated with jME3 [31] through the *com.jme3.niftygui* package.

2.10 Client/Server

The Client/Server Network architecture involves multiple clients connecting to a single, central Server. Usually the file server on a Client/Server network is a high capacity, high speed computer with a large hard disk capacity. This kind of technology is implemented in most of the professional 3D games, in particular the Massively Multi-player Online Games (MMOG) [7], in which the huge number of Clients connected requires a high-computational powered server architecture responsible to manage all the game state computations. The drawback of this model is its non-scalability: as the number of participants in a Virtual Environment increases, the server works as a bottleneck. Even if additional Servers are used, the delay due to additional communication overhead in servers is inevitable [35].

2.11 Peer-To-Peer

Peer-To-Peer networks involve two or more computers sharing individual resources. These resources are available to every computer in the network. Each computer acts as both the Client and the Server, meaning that all the computers on the network are equal, not requiring any additional coordination entity (such a central Server) and not delaying transfers by routing via Server entities. Recently, Virtual Environment middlewares such as ATLAS [35], Mediator [20] and Hydra [10] implemented a Peer-To-Peer architecture in order to face the scalability issues related to the Client/Server architecture.

2.12 JXTA

Juxtapose (JXTA) [54] is an Open Source Peer-To-Peer protocol specification working through a set of XML messages which allow any device connected to exchange messages and collaborate independently of the underlying network topology. Implementations of JXTA [54] are available for Java SE, C/C++, C# and Java ME. The peculiarity of JXTA [54] is the possibility to create a virtual overlay network which allows a Peer to interact with other Peers even when some of them are behind NATs and firewalls or use different network transports.

As explained in [68], the JXTA [54] Architecture is made of three logical layers:

- **Platform Layer**: it is the base of JXTA [54] and contains the implementation of the minimal and essential functionalities required to perform Peer-To-Peer networking. This layer is also know as the core layer.

- **Services Layer**: it contains additional services that are not absolutely necessary for a Peer-To-Peer system to operate, but which might be useful: searching and indexing, storage systems, file sharing, distributed file systems, resource aggregation and renting, protocol translation, authentication, PKI (Public Key Infrastructure).

- **Applications Layer**: it includes implementation of integrated applications: Peer-To-Peer instant messaging, document and resource sharing, content management and delivery, Peer-To-Peer email systems, distributed auction systems.

In order to know the advantages and the disadvantages of the JXTA [54] technology, it is interesting to read the performance study written by Halepovic and Deters [27].

2.13 DB4O

DB4O (DataBase FOR Objects) [67] is an embeddable Open Source object database for Java [51] and .NET developers. It is developed, commercially licensed and supported by Versant. DB4O [67] is written in Java and .NET and provides the respective APIs. It can run on any OS that supports Java [51] or .NET. It is offered under multiple licenses, including the GNU General Public License (GPL), the DB4O Opensource Compatibility License (dOCL), and a commercial license for use in proprietary software.

DB4O [67] represents an object-oriented database model. One of its main goals is to provide an easy and native interface to persistence for object-oriented programming languages. Development with DB4O [67] database does not require a separate data model creation, the application's class model defines the structure of the data in DB4O [67] database. DB4O [67] aims to avoid the object/relational impedance mismatch by eliminating the relational layer from a software project.

A very interesting project [13] uses DB4O [67] for universal storage, primarily for SceneNode persistence and Linq queries.

2.14 Ozone

Ozone [19] is a fully featured, object-oriented database management system implemented in Java [51] and distributed under the LGPL Open Source license. The

Ozone [19] project aims to evolve a database system that allows developers to build pure object-oriented systems. Ozone [19] does not depend on any back-end database to actually save objects. It contains its own clustered storage and cache system to handle persistent Java [51] objects. Ozone [19] includes a fully W3C compliant DOM implementation that allows to store XML data. It is possible to use any XML tool to provide and access these data. Support classes for Apache Xerces-J and Xalan-J are included.

2.15 C++

C++ [12] is a statically typed, free-form, multi-paradigm, compiled, general purpose programming language. Today, because it is object-oriented and compiles to binary, the most popular game development language is C++ [12]. Because of this, the majority of commercial computer and video games are written primarily in C++ [12].

2.16 Java

Java [51] is a platform-independent, object oriented language that aims to the development of applications to run in network environments and the Internet. Nowadays Java [51] can be executed by most of the existing browsers [34, 47] through the Java Web Start technology. Most of the Virtual Environment (VE) middlewares [47, 45, 36, 33] make use of Java [51] as the primary language. Nevertheless Java [51] has always been considered slower than other development languages such as C++, nowadays the performance offered by Java [51] when a 3D accelerated graphics card is used is fully comparable to C++. Ingles [30] investigated Java's performance for game programming, showing that by the introduction of the full-screen exclusive mode in J2SE version 1.4 programs using Java [51] are able to access graphics hardware more directly. The addition of full-screen exclusive mode alone has increased Java's graphics performance speed by about three times.

2.17 Lombok

Project Lombok [75] is a small Java [51] library that can be used to reduce the amount of Java [51] boilerplate code that is commonly written for Java [51] classes. Project Lombok [75] does this code reduction via annotations that can be added to the Java [51] class replacing desired common methods. It is possible to annotate any field with *@Getter* and/or *@Setter*, to let Lombok [75] generate the default accessors automatically, or with *@Delegate* in order to generate delegate methods that forward the call to the given field; any class definition may be annotated with *@ToString* to let Lombok [75] generate an implementation of the *toString()* method, or with *@EqualsAndHashCode* in order to obtain the Lombok [75] implementations of the *equals(Object other)* and *hashCode()* methods, or with *@NoArgsConstructor* to generate a constructor with no parameters.

Chapter 3

Related work

Over the last decade, research efforts in the Virtual Environment middleware industry were targeted at improvement of scalability, persistence and responsiveness capabilities, while much less attempts [44] have been aimed at addressing the flexibility, maintainability and extensibility requirements in contemporary Virtual Environment platforms.

ATLAS [35] is a Distributed Virtual Environment (DVE) system that allows users on a network to interact with each other by sharing a common view of their states. This system is designed with the scalability requirement in mind. Several techniques are adopted, such as the subdivision of the virtual world into several logical regions in order to reduce message exchange, or a prediction-based concurrency control, implemented in order to allow real-time interaction for users. Interest management and scalability requirement are faced also in the Message Oriented Middleware [39], through which the traditional interest management techniques such as the region-based technique and the aura-based technique are improved by a new predictive interest management scheme. Move [23] and the Joint Hierarchical Nodes based User Management (JoHNUM) are another efforts in creating a middleware that is completely scalable, consistent and high-performance-based, with particular attention given respectively to data extraction and region-based interest management. Reducing bandwidth use and latency optimization are other design problems in the construction of a Virtual Environment: Fabre [18] faced this kind of issues by creating a new Virtual Environment middleware based on VRML, X3D and Java [51] languages. The new important feature introduced by this framework is the possibility to embody in the Virtual Environment autonomous creatures, called mobile agents, whose behaviour is easily defined through the middleware. Also Quax [58] addressed the use of autonomous avatars inside a Virtual Environment. Through the Architecture for Large Scale Virtual Interactive Communities (ALVIC) framework, Quax used the autonomous avatars to simulate a large-scale multi-user networked environment, providing a method for scalability testing, eliminating the need for large numbers of human users and obtaining accurate results by only using a limited number of computers. Other researches [17] focus instead on the performance requirement, providing enhancement techniques related to the synchronous communication, needed in a Virtual Environment in order to obtain a good quality of the experience.

The majority of the Virtual Environment systems described above tried to solve some of the distributed problems inherent the design of Virtual environments, but they were based on monolithic architectures, making difficult maintenance and software reuse. Projects are often developed from scratch and involve a great deal of programming without reusing building blocks. OpenPING [44], a reflective middleware for the construction of adaptive networked game applications, is one of the first Virtual Environments middlewares which aims to reach the flexibility, maintainability and extensibility requirements by providing an architecture implementation entirely based on modules, such as the Concurrency module, the Replication module, the Interest Management module, the Persistence module, the Consistency module and the Event Channelling module. The separation of the functionalities of the framework provides the powerful possibility to extend the framework itself, by adding new functionalities or by replacing the existing one with new implementations. The developer benefits of this subdivision because he does not have to ”reinvent the wheel” each time he wants to create a new Virtual World, but he is free to reuse the modules offered by OpenPING and other modular Virtual Environments middlewares, in order to implement a specific functionality in his project. Contigra (Component OrieNted Three dimensional Interactive GRaphical Applications) [14] is another effort to build a component-based architecture on the basis of X3D and XML, designed for the construction of web-based, desktop Virtual Reality applications and 3D Virtual Environments. The Contigra project pays great attention to the concepts of reuse, by providing a methodological approach largely independent of implementation issues. Also Java Adaptive Dynamic Environment (JADE) [46] and Mammoth [33] supply a full modularity function, adding a proper Module Manager in order to easily administrate the modules implemented in the framework. Recently, the Open Wonderland project [47] has attracted attention in the Virtual Environment middleware industry. Open Wonderland is a 100% Java [51] Open Source toolkit for creating collaborative 3D worlds. One of the best features that led Open Wonderland to success is just the possibility to extend any part of the system and add functionality by creating modules, the Wonderland version of plugins. Oliveira [45] presented the Virtual Environment System Layered Object Model (VESLOM), providing a complete approach to design fully modular Virtual Environment frameworks. The aim underlying VESLOM [45] is to reduce the complexity related to the design of a Virtual Environment middleware by dividing it into different layers. Starting from the core functionality, provided by the so-called Universal Platform, the developer selects the appropriate components for his application, such as the Networking layer, the Middleware layer and the Application layer. The VESLOM [45] was successfully used in the creation of the Java Adaptive Dynamic Environment (JADE) [45], a fully modular Virtual Environment system.

Other Virtual Environments middlewares focus on the collaborative aspect: these middlewares are called Collaborative Virtual Environments (CVE) frameworks. According to the definition specified in [34], a CVE is a multi-user, Collaborative Virtual Environment that runs on the Internet. Several projects involve the creation of specific tools through which it is possible to create Virtual Worlds. CVE-VM [34] is a Collaborative Virtual Environment tool, created aiming to support the teaching/learning in Brazilian schools. The CVE-VM implementation uses VRML 2.0 and Java [51] languages, integrated by the EAI (External Authoring Interface). The CVE-VM system is developed according to the Client/Server model, in which the Server is responsible for managing the communication among the users and maintaining the consistency of the database, allowing the clients to use a browser to navigate and manipulate objects in the Virtual World. Specific efforts [57] have been made in order to define several collaborative manipulation techniques, through which multiple users simultaneously can manipulate an object in a Virtual Environment. CVEs have been created also for Data Knowledge Extraction: this is the case of the LiveNet system [4], developed at the University of Technology, Sidney. Virtual collaboration systems do not provide any support for data collection aimed at knowledge discovery while the LiveNet [4] framework embeds knowledge discovery by applying data mining algorithms through which it is possible to discover useful patterns in the data.

An other important aspect of the Virtual Environment middlewares is the choice of the network architecture. Most of the professional Virtual Environment tools adopt the Client/Server architecture, while recently Peer-To-Peer architecture was taken in consideration by more and more projects [35, 20, 10], or even hybrid architectures [35]. In Peer-To-Peer based Networked Virtual Environments, system and data management is distributed among all the participating users. By sharing users' resources, Peer-To-Peer architectures achieve high scalability in a cost-effective manner. In order to reduce the amount of message traffic, several techniques have been proposed, such as the subdivision of the virtual world into portions commonly known as Area of Interest (AOI), allowing the user to interact only with entities within his AOI, or the Delaunay Triangulation [8], a technique for dramatically decreasing maintenance overhead by reducing the number of connection changes due to users' insertion and movement.

Recently, Virtual Environment systems were affected by the growing interest in the use of scripting languages. Powerful scripting languages improve application performance while making the development more efficient [72, 60]. These languages allow developers to easily specify how an object or character is supposed to behave, without worrying about how to integrate this behaviour into the application itself. User created content is another reason for Virtual Environment applications to support scripting. Virtual Worlds such as Second Life [61] have made player scripting a common topic of conversation. Besides, scripting allow players to modify application behaviour without having access to the code base. They provide a sandbox that - unlike a traditional programming language - limits the types of behaviour the player can introduce.

Our aim is to provide a new Java-based Multi-user Modular Networked Virtual Environment framework, the Java Interactive Virtual Environment System (JIVES). JIVES is completely Open Source and released under the GNU General Public License 3.0 (GPLv3) [22]. All the technologies employed in the creation of the JIVES framework are Open Source based. JIVES has been designed following the VESLOM [45] approach in order to make it modular and extensible.

JIVES is composed by a Universal Platform, which provides the core functionalities of the framework, a Networking layer and a Presentation layer. According to the VESLOM [45] approach, the Universal Platform represents the kernel of the middleware. JIVES is provided by default with a Presentation Layer and a Networking Layer, the first implemented by using the jMonkeyEngine [31], the latter by using the JXTA technology [54], facing in that way the scalability problem related to Client/Server architectures. The modularity offered by JIVES allows the developer to implement any other kind of Presentation Module and Networking Module, using different technologies than the ones implemented by default. The extensibility offered by JIVES lets the developer fully personalize some of the features of the framework: in addition to the actions and events implemented in the framework, new customized actions and events can be defined and used in the development of a new Virtual Environment. Because of the choice of a Peer-To-Peer architecture, the JIVES framework is not properly suitable for the creation of Massively Multi-player Online Games, although this could still be done.

The Virtual World model adopted by JIVES follows the one proposed by Menchaca [36]. The Virtual Worlds that can be created by using our framework are composed by three main elements: Individuals, Artefacts and Decorations. Menchaca [36] defines the Individuals as users' avatars that interact within the Virtual World. JIVES allows the developer to define the actions that users are able to perform within the Virtual Environment. Artefacts are elements, Individuals can interact with, while Decorations are static objects (or animated with deterministic time or event based behaviour) that are visible within the Virtual World but do not have collaboration interfaces, as Menchaca [36] explains in his study. Unlike projects mentioned before which face specific Virtual Environment design issues, JIVES wants to be as simple and general as possible, allowing the developer to build a new Virtual World in a fast and simple way. In order to meet this requirement, JIVES provides its own scripting language, called JiveScript. JiveScript is a structured scripted Java [51] language, based on the scripting API *javax.script* [52], package available in the Java SE 6 platform. Through JiveScript the developer can create efficiently a new 3D Virtual Environment, specifying the whole logic of the application, the virtual objects and their behaviours. Being JiveScript a structured language, the developer can even make use of a specific application to write the code, instead of writing it directly. In order to facilitate the developer, we published a NetBeans plugin [15] so that it is possible to take advantage of the instant code auto completion feature. As mentioned by White in [72], the choice of adopting a scripting language provides an additional layer of security: JiveScript can be executed in a sandbox in which the developer is free to test his application changing several features and properties of the entities by means of scripting. When the script is complete and not modifiable anymore, the developer can block the sandbox mode in order to avoid that other users change the script, even minimally.

Chapter 4

Motivations and targets

Building a Virtual Environment is an extremely challenging and resource consuming task. Indeed many of the discussed related work are thesis per se, involving on the whole years of research and teams composed by several people that cooperate in order to create a commercially exploitable software. Our approach to the problem cannot be seen as an universal solution, fully working and efficiently performing; this would require much higher costs in terms of time and human and hardware resources than we could afford in the immediate.

The main objective of this thesis can be found in achieving a general, simple and versatile solution. We perceived this objective as a design challenge and put all our effort in writing a smart and working implementation, while investigating many of the aspects that commonly concern this kind of development. We were not interested in creating a 3D Virtual Environment in itself, but to create something more: an experimental system that would help others create their own 3D applications. The JIVES framework aims to be a starting point in the direction of creating a Virtual Environment framework that includes all the features needed in the development of such applications. By analysing the literature of the last decade, we realized that a lot of problems have plagued the development of Virtual Environment systems. Starting from the ashes of an old project of 2008 [16], we decided to completely redesign the JIVES framework in order to obtain a new system that keeps pace with the times and takes into account the problems that have afflicted the development of Virtual Environment systems in recent years, such as scalability, extensibility and ease of use. Because of this, the JIVES framework has been designed modular regarding the Presentation and Networking layers in order to face the extensibility issue and the currently provided network module is based on a Peer-To-Peer architecture in order to take into account scalability issues related to a client-server architecture. The modularity of the JIVES framework would let the developer implement a different network module, maybe Client/Server based or a Hybrid architecture, in order to meet all the requirements of the application in development. In addition, one of the most important reasons that led us to the realization of this new version of the JIVES framework is the fact that, by analysing the existing products on the market, we realized most of them are aimed at very expert users. It is precisely for this reason that we decided to introduce in the realization of JIVES the possibility of using a scripting language, the JiveScript, through which also those who are not expert in programming have the possibility to create their own 3D application using our framework. JIVES lets the developer choose to code an application directly in the Java [51] language, or by using our scripting language. The best choice is obviously somewhere in between: by coding in the Java [51] language, the developer can create new Jives modules, called Implementors, in which there is the actual implementation of some features needed by his application, such as a Presentation implementor and a Networking implementor; while the whole logic of the application can be coded directly in JiveScript. In that way, JIVES benefits from a large re-usability: the next time that it is necessary to develop an application that should handle similar virtual worlds, for example any art gallery, the developer will need to write only the application logic using JiveScript, since the art gallery visualization code has been already developed in the Presentation Implementor.

Because of the over-abundance of projects related to Virtual Environments design issues, such as the scalability, consistency and persistency requirements, JIVES is not meant to be specialized in the optimization of any particular requirement (although this kind of optimizations could be added in the future), but differs for other reasons. Unlike the existing Virtual Environments frameworks, JIVES offers a fully integrated Item Inventory Management System, through which the developer can manage all the items of the virtual scene, specifying their implementation and behaviour. Through JIVES, it is possible to create a Virtual World in which users have their own inventory, exchange items with each others or with Non-Playing characters and use items to interact with the scene. This feature allows JIVES to be particularly suitable for the creation of Inventory-based games, such adventure games or simulations, but nothing prevents the developer from creating any other kind of Virtual Environment or 3D application, by using our framework. The additional feature offered by JIVES is the possibility to combine different items together in a smooth and interactive way: through the JiveScript language, the developer can define the relations between the Inventory items allowing the creation of new items from existing ones. The novelty introduced by JIVES is the possibility to define for each virtual item a list of its possible "hotspots", or attachment points, through which the item can be combined with another one. In that way the developer is able to define more complex and meaningful combinations than those usually possible in the current Virtual Environments middlewares.

We want to offer a product extensible and renewable by the addition of new modules and customizations that allows the creation of new 3D Virtual Environments. The freedom that we offer to the JIVES user is such that he can develop any type of 3D application using our framework. JIVES does not limit in any way the creativity of the developer and it can be even used to create multi-access interactive web virtual applications. The range of applications that can be developed using JIVES is very large: from Inventory-based games to training simulations, such as simulations aimed to train the user in critical situations, like a fire drill evacuation, or simulations used to prepare for a hostile environment, the same way a survival course does, to collaborative E-Learning Virtual Environments, such as interactive 3D class rooms, or professional working environments that can support collaborative projects, or virtual cities, museums, art galleries and labs. By using JIVES the developer can create something that offers more than some of the applications already existing on the market: for example, it is even possible to improve applications like Google Business View [25], a tool that allows you to view the inside of stores through a 360-degree imagery based on the Google Street View Technology [26], by adding an additional layer of Immersivity: the Interactivity. By using JIVES, the developer can create a specific application to let the user to not just visit the online store but also interact directly with it by exploring the shop and examining the products.

The Virtual Worlds created by using our framework must have the element that we believe the most important one in a Virtual Environment application: an enjoyable participatory experience. So that, the implementation of the Virtual Environment must have enough performance to allow for a smooth virtual experience, and the application states perceived by the participants must be similar enough to not give an unfair advantage to any of the users. The choice of a Peer-To-Peer architecture, used to implement the Networking layer, aims to have at most a hundred users simultaneously connected in the same Virtual Environment: that number of participants is more than enough to achieve a fully enjoyable online experience. There are a lot of studies that prove the excellent scalability of this kind of network architecture: in particular, Hu and Chen [29] have successfully tested a Peer-To-Peer network consisting of more than 2000 nodes.

Chapter 5

Design and Implementation

5.1 Design

The design phase of the JIVES framework is completely based on the Virtual Environment System Layered Object Model (VESLOM) approach [45], depicted in Figure 5.1.

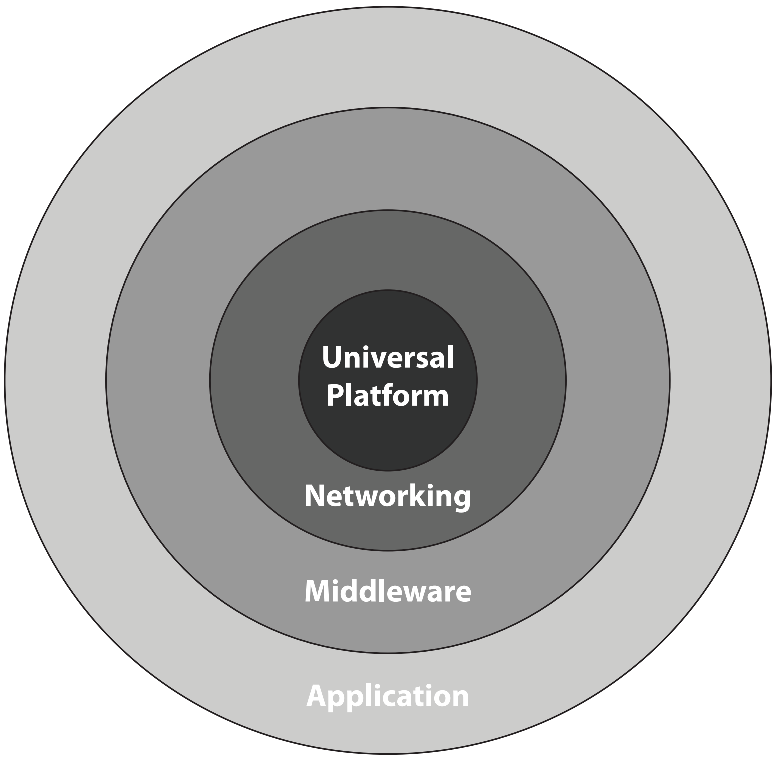


Figure 5.1: Virtual Environment System Layered Object Model

The choice to follow the VESLOM approach is due to the will to avoid the monolithic design used by the Virtual Environment systems of the last decade, in which each time a new set of requirements are specified, a new Virtual Environment system is designed and implemented. At the contrary, the VESLOM approach offers a novel methodology to develop Virtual Environment systems, based on a strong layered component design. The approach underlying VESLOM is to reduce the complexity in creating a new Virtual Environment system by dividing this complexity into different layers: each of them provides a specific functionality of the overall system.

As described in the VESLOM approach [45], the core functionality of JIVES lies in the so-called Universal Platform module. There is no need to adopt an entire system but only the Universal Platform and select the appropriate components, or build new ones, to have a fully working Virtual Environment middleware. In particular, JIVES Universal Platform has been designed with specific functionalities in mind: in addition to the JIVES Core, which represents the kernel of our framework, the Universal Platform includes the Persistence Layer, the Actions and Events Management System, the Dialogues Manager and finally the Bag Graph (BG). The Universal Platform has been designed in this way to let the developer immediately start writing his own product, since the JIVES Universal Platform includes all the main features of a Virtual Environment.

According to the VESLOM approach [45], the modules we provide in addition to the Universal Platform are the Networking Layer, which has the responsibility to manage all the networking functionality, and all the components encompassed in the Middleware Layer. The Networking Layer implemented by default in the JIVES framework is based on a Peer-To-Peer technology, the JXTA[54] technology, and in particular its Java [51] version, JXSE. The decision to choose a Peer-To-Peer technology is dictated by the will to offer to the JIVES developer the possibility to create its own Virtual Environment, that can be used also online at no cost.

The Middleware layer includes all the remaining components necessary to build a Virtual Environment. In particular, the Middleware Layer includes the Presentation module and all the possible extensions introduced by the JIVES developer. The Presentation layer, also called Engine Implementor, offered by default in the JIVES framework, is based on the jMonkeyEngine [31] technology, and provides the possibility to create a completely 3D Virtual Environment Application. The choice of jMonkeyEngine [31] as the Presentation technology of the JIVES framework is dictated by the fact that the jMonkeyEngine [31] is a completely free and Open-Source Java-based game engine: this means that it offers a lot of features that can be useful to JIVES, such as collisions, particle systems, shaders, terrain system and renderer abstractions. An additional implementation of the Presentation module has been done in order to have all the functionalities of the JIVES framework running also in a textual mode: through the Shell Implementor the user can create a fully working textual Virtual Environment. In addition to the Presentation module, the Middleware layer includes also all the possible extensions introduced by the JIVES developers, such as new events and actions. In that way, JIVES can be expanded offering new customizations that also other developers can use in writing their Virtual Environment applications.

The VESLOM approach [45] defines the Application layer as a representation of the final Virtual Environment system that the user interacts with: it contains all the components that are tightly coupled with the specific functionality and instantiation of the Virtual Environment. For this reason, the Application layer of the JIVES framework has been designed having in mind the idea to offer a new smooth and interactive way to write 3D Virtual Environment applications: this is possible through the use of the JiveScript scripting language. JiveScript has been introduced in the JIVES framework to let the developer define efficiently his own 3D Virtual Environment, specifying the whole logic of the application, the virtual objects, their relations and behaviors.

The Class Diagram of the JIVES project is fully reported in Appendix A.

It is important to stress that the JIVES framework has been designed to be independent of the reference implementation due to the use of the Java [51] programming language: Java [51] programs are platform independent and can be executed in every operating system through the use of a Java Virtual Machine. In order to speed up the writing of the boilerplate code and ease its maintainability, JIVES has been developed using the Lombok [75] technology, particularly for the accessor and delegate methods generation.

5.1.1 Inventory Data Structure

A fundamental aspect of the proposed design consists in the fact that every character has his own (eventually more than one) inventory, also referred to as ”bag”, for brevity. The character is able to switch between any of his bags, extract items to use in the environment or to exchange with others, and he can perform other actions on those items like examining them or combining them. Combination plays a major role in this interaction. We introduce a new paradigm where the end user must decide not only which items to combine, but (eventually) in which way to combine them. In order to do so, the underlying data structure of a bag, known as the Bag Graph, not only must carry the information about combinations, but must also define if a combination is accepted; thus, we introduce the concept of HotSpot. HotSpots are physical locations or descriptive attributes (or eventually other entities) that are attached to bag items and the combination of two items is defined by mean of matching an hotspot from the first item with an HotSpot from the second one.

Figure 5.2 shows a simple Bag Graph that defines three bag items (a nylon wire, a wooden stick and a hook) and the necessary combinations to obtain a fishing rod. Please note that while bag items are nodes of the graph, HotSpots are the arcs. A node can have many outgoing arcs, but either zero or two incoming

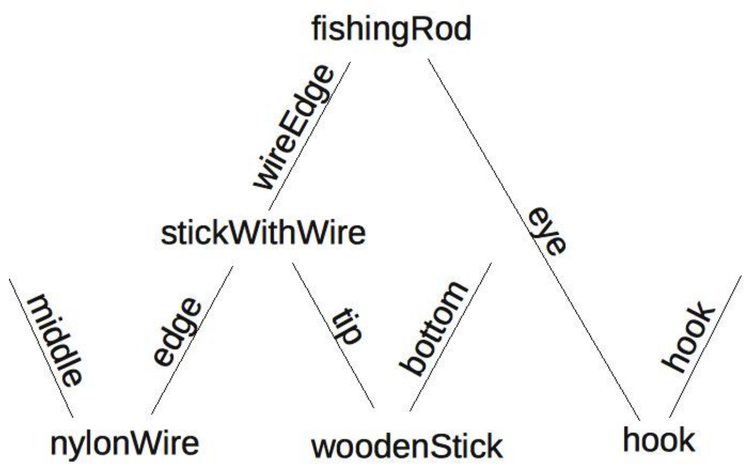


Figure 5.2: Bag Graph Example

arcs; in the case it has incoming arcs, those identify the two parent nodes and the previously mentioned node is the result of their combination. Also, some arcs are dangling; they are useless in terms of combination definition, but yet they are present so that the puzzle solver has to include them in his/her reasoning. Each arc in Figure 5.2 is labelled with a textual attribute describing which part of the object is physically interested in the combination (for example, the wire edge with the stick tip derive a stick with an attached wire), and also nodes are represented by a textual description. But this is only a possible model of this bag graph, because in a visual environment, for instance, the nodes would be 3D models while HotSpots would be predefined volumes located in the models space.

5.1.2 Event Driven Architecture

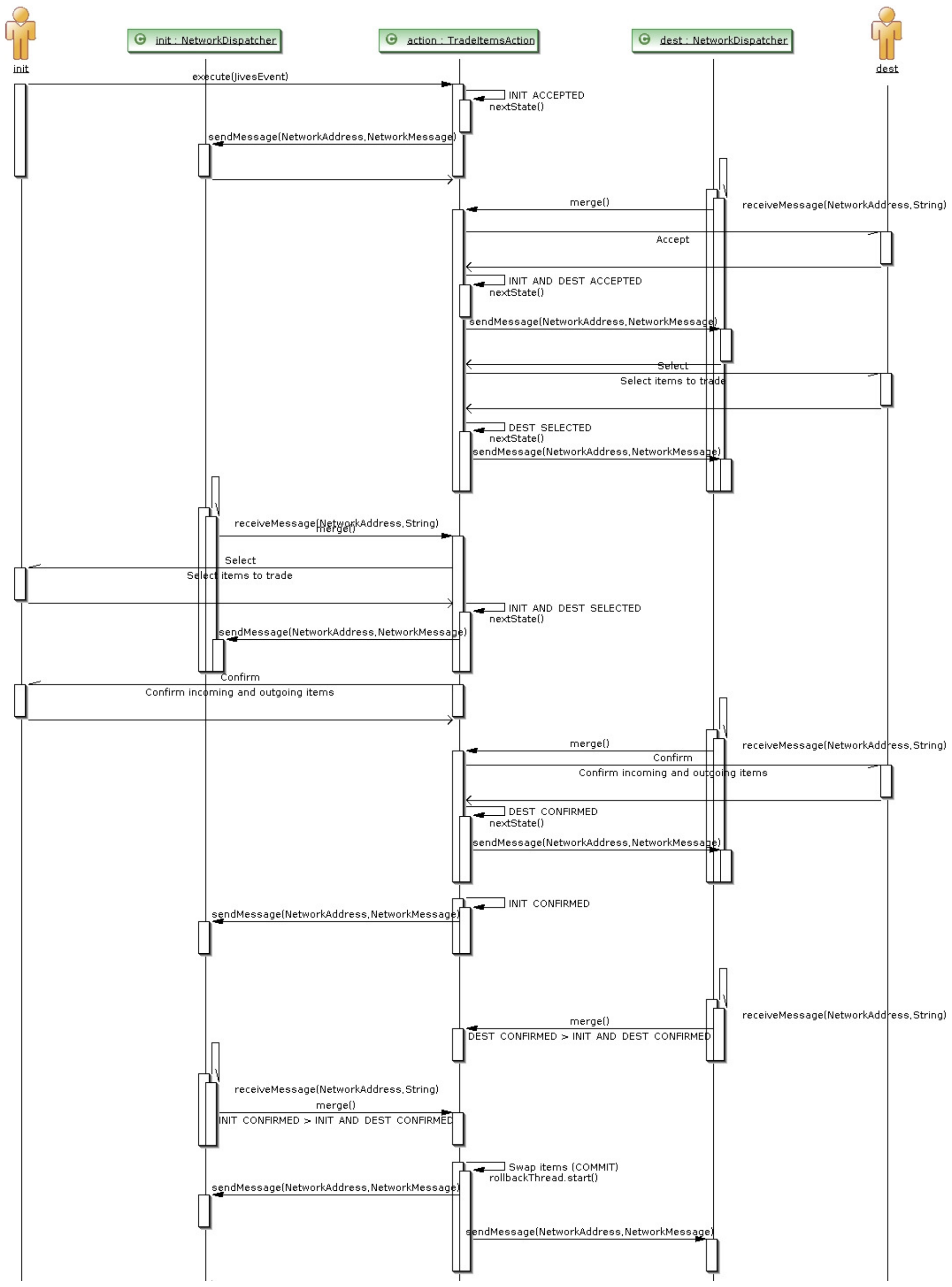
The struggle for generality implies that the system must provide a certain degree of flexibility toward application-specific behaviours, as long as it is not possible nor convenient to foresee each and every possible scenario directly in the framework design. This is the main reason why we introduce the events, the most basic feature of a work-flow management system [74]. Actions are not simple tasks or routines that get executed unconditionally and immediately after an activation has been issued, but events triggered by the activation of any activable entity instead.

The advantage that this design bring is immediately comprehensible: events can be intercepted, delayed, conditionally evaluated and consumed, concatenated, inhibited perhaps, while the correspondent action is executed, or trashed, accordingly. The top level application can exploit the event based approach to implement its own logic upon every activation, both issued by the end user or by the system itself, without breaking the encapsulation of the lower levels and remaining completely agnostic about the internal handling mechanisms. The event driven architecture [37] is described further on in section 5.2.1.

5.1.3 ***TradeItemsAction*** Protocol

This paragraph explains in detail one of the possible actions designed to be performed in the environment. Specifically, the trade action allows to exchange items between inventories and has transactional behaviour: the whole operation has to be either committed on both sides, or rolled back on both sides, eventually compensating the temporary changes that has been introduced on the inventories. Even though "trading" is not the only possible transaction to be defined in the JIVES Environment, it's provided by default among with some other basic actions, thus its practical study is presented here as a sequence diagram in Figure 5.3.

The steps before transaction goes into INIT_AND_DEST_CONFIRMED are handshaking steps and allow both users to select which items to give from their bag and to see which item will be received from the counterpart. Note that each time a transaction is dispatched by the *NetworkDispatcher* class, the local copy of the same transaction is merged with the remote one, so that its internal state can be synchronized. When the transaction goes in INIT_AND_DEST_CONFIRMED state on the two peers, symmetrically, the commit is performed and a roll-back thread starts. The roll-back thread has a time-out of ten seconds, at the end of which the transaction is aborted and the previous commit is compensated. During


init
init : NetworkDispatcher
action : TradeItemsAction
dest : NetworkDispatcher
dest
execute(JivesEvent)
INIT ACCEPTED
nextState()
sendMessage(NetworkAddress,NetworkMessage)
merge()
receiveMessage(NetworkAddress,String)
Accept
INIT AND DEST ACCEPTED
nextState()
sendMessage(NetworkAddress,NetworkMessage)
Select
Select items to trade
DEST SELECTED
nextState()
sendMessage(NetworkAddress,NetworkMessage)
receiveMessage(NetworkAddress,String)
merge()
Select
Select items to trade
INIT AND DEST SELECTED
nextState()
sendMessage(NetworkAddress,NetworkMessage)
Confirm
Confirm incoming and outgoing items
merge()
receiveMessage(NetworkAddress,String)
Confirm
Confirm incoming and outgoing items
DEST CONFIRMED
nextState()
sendMessage(NetworkAddress,NetworkMessage)
INIT CONFIRMED
sendMessage(NetworkAddress,NetworkMessage)
receiveMessage(NetworkAddress,String)
merge()
DEST CONFIRMED > INIT AND DEST CONFIRMED
receiveMessage(NetworkAddress,String)
merge()
INIT CONFIRMED > INIT AND DEST CONFIRMED
Swap items (COMMIT)
rollbackThread.start()
sendMessage(NetworkAddress,NetworkMessage)
sendMessage(NetworkAddress,NetworkMessage)

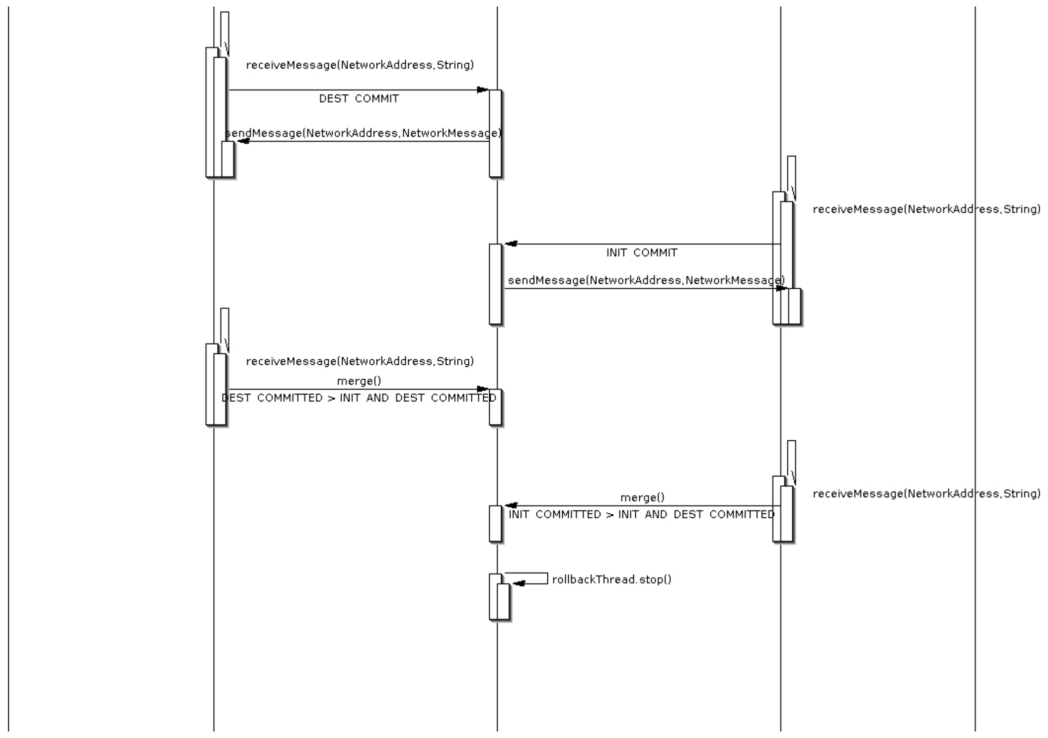

receiveMessage(NetworkAddress,String)
DEST COMMIT
sendMessage(NetworkAddress,NetworkMessage)
receiveMessage(NetworkAddress,String)
INIT COMMIT
sendMessage(NetworkAddress,NetworkMessage)
receiveMessage(NetworkAddress,String)
merge()
DEST COMMITTED > INIT AND DEST COMMITTED
merge()
INIT COMMITTED > INIT AND DEST COMMITTED
receiveMessage(NetworkAddress,String)
rollbackThread.stop()

Figure 5.3: TradeItemsAction Commit

this period of time, both peers must reach the INIT_AND_DEST_COMMITTED state, which is reachable if:

- The initiator committed and received the confirmation of the destination commit.
- The destination committed and received the confirmation of the initiator commit.

If a peer commits and doesn't receive the confirmation from the counterpart within the roll-back thread time-out, it sends an abort message and it compensate its commit. At that point, when the counterpart receives the abort message, it compensate its commit too, even if it had already gone into INIT_AND_DEST_COMMITTED state. The protocol works using TCP/IP so that the messages are always delivered, and the only issue is the time that can intercur from packet send to packet receive, which depends mainly on network load. In case that, due to the high network load, the confirmation message is not received before the expiration of the roll-back countdown, the transaction is simply aborted.

5.1.4 Networking Layer Architecture

The Network Architecture is essentially the one offered by the JXTA [54] technology. The *JXSEImplementor* represents the JIVES Network Layer and implements all the network functionalities of the framework. While designing the Implementor, only the Core Layer of the three Layers that make up the JXTA [54] architecture has been taken in consideration. There is no need to adopt also the Services Layer and the Application Layer, because JIVES requires only the minimal and essential primitives that are common to Peer-To-Peer Networking. This includes the key mechanisms for a Peer-To-Peer application such as Peers discovery, communication transports, creation of Peers and Peer Groups; all services belonging to the JXTA [54] Core Layer.

The JXTA [54] Architecture, as specified in [68], provides a subdivision of the Peers into three main categories:

- **Minimal-Edge Peers**: Peers that implement only the required core JXTA [54] services.

- **Full-Edge Peers**: Peers that implement all the core and standard JXTA [54] services and can participate in all of the JXTA [54] protocols.

- **Super-Peers**: Peers that implement and provision resources to support the deployment and operation of a JXTA [54] network. There are three Super-Peer functions:

  Relay: used to store and forward messages between Peers that do not have the direct connectivity because of firewall or NAT.

  Rendezvous: maintains global advertisement indexes and assist Peers with advertisement searches. Also handles message broadcasting.

Proxy: used by Minimal-Edge Peers to get access to all the JXTA [54] network functionalities.

JXSEImplementor takes possession of this Peers subdivision with some modifications: in the *JXSEImplementor* there are only two kinds of Peers, the Full-Edge Peers and the Rendezvous/Relay Peers. The JIVES Full-Edge Peers can send and receive messages and cache advertisements. They reply to discovery requests with information found in their cached advertisements, but do not forward any discovery request. The JIVES Rendezvous/Relay Peers are Super-Peers that have been designed to have both the functionalities of the Rendezvous Peer and the Relay Peer, in such a way that a NATed Peer has to connect to only one Super-Peer (the JIVES Rendezvous/Relay Peer) instead of two different Super-Peers (a Rendezvous and a Relay Peer) in order to have a fully working Internet connection. Each JIVES Peer, that is either a simple Peer or a Rendezvous/Relay Peer, exposes a public address and a private address. This distinction is needed only when using an IPv4 connection, and it is not necessary when using an IPv6 connection. The public address represents the IPv4 address provided by the router used to manage the connection, while the private address is the LAN address of the network managed by the router.

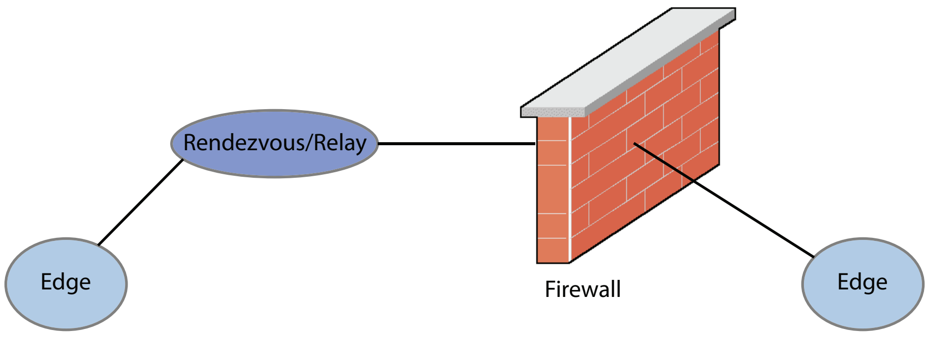


Figure 5.4: JIVES Networking Layer Architecture: *JXSEImplementor*

When JIVES runs a Virtual Environment application in the LAN mode, only the private address is needed to connect Peers belonging the same LAN, while in the Internet mode both addresses are needed: the public address is used to reach the router, the private address is used to reach the Peer behind the router. If a user has a direct connection to Internet (without passing through a router) or when an IPv6 address is used, there is no need of the distinction between the public address and the private address: they coincide. The basic functioning of the *JXSEImplementor* is shown in Figure 5.4. The *JXSEImplementor* does not take into consideration the JXTA [54] Proxy Peers, because it has been designed to support Proxy connections in a different way: in Java [51], there is the need to set only some system properties, as shown in Code Snippet 5.1.

Code Snippet 5.1: Proxy connection

```
// Reading networkConfiguration
XMLConfigParser.readProxyConfiguration();

// Enabling the properties used for Proxy support
System.getProperties().put("proxySet", "true");
System.getProperties().put("proxyHost", XMLConfigParser.proxyHost);
System.getProperties().put("proxyPort", XMLConfigParser.proxyPort);

// Removing inactive Rendezvous
queryInactive();

// Querying the Rendezvous file
queryFile(scriptName, scriptMD5);

URL url = new URL(remote_url);
URLConnection urlConn = url.openConnection();

// Entering the Proxy username and password
String password = XMLConfigParser.proxyUsername + ":"
+ XMLConfigParser.proxyPassword;

// base64 encoding of the password
String encoded = Base64.encodeBase64String(password.getBytes());

// Setting up the connection
urlConn.setRequestProperty("Proxy-Authorization", encoded);
```

JXTA [54] provides a Core set of services:

- **EndPoint Service**: used to send and receive messages between Peers.
- **Resolver Service**: used to send generic query requests to other Peers.

In addition to the Core services JXTA [54] defines additional standard services:

- **Discovery Service**: used by Peers to search for Peers resources.
- **Membership Service**: used by Peers to securely establish identities and trust within a Peer Group.
- **Access Service**: used to validate requests made by one Peer to another.
- **Pipe Service**: used to create and manage pipe connections between the Peers.
- **Monitoring Service**: used to allow one Peer to monitor other members of the same Peer Group.

JXSEImplementor takes possession of all the Core set of services, while with respect to the Standard set of services only the *Discovery Service* and the *Pipe Service* are used in the JIVES Networking Layer Architecture.

With regard to the exchange of messages between Peers, JXTA provides this communication through communication channels called pipes. In particular, JXTA [54] defines three types of pipes:

- **Point-To-Point Pipes**: connect exactly two endpoints together, an input pipe on one Peer receives messages sent from the output pipe of another Peer.
- **Propagate Pipes**: connect one output pipe to multiple input pipes; messages flow from the output pipe, the propagation source, into the input pipes.
- **Secure Unicast Pipes**: is a Point-To-Point Pipe that provides a secure and reliable communication channel.

JXSEImplementor has been designed to use a single pipe communication channel for each JIVES Virtual Environment application: the choice to support a fully broadcast communication has led to choose the *Propagate Pipe* as the JIVES communication channel in order to propagate a message sent by a Peer running a given application to all the Peers running the same application.

In JXTA [54], all the network resources, such as Peers, Peer Groups, Pipes and Services, are represented as *advertisements*. *Advertisements* are language-neutral meta-data structures represented as XML documents. The JXTA [54] protocols define eight different types of *advertisements*:

- **Peer Advertisement**: describes the Peer's resources, such as the name and the ID.
- **Peer Group Advertisement**: describes Peer Group specific resources.
- **Pipe Advertisement**: describes a pipe communication channel and it is used by the Pipe Service to create the associated input and output pipe endpoints.
- **Module Class Advertisement**: describes a module class, a low-level JXTA [54] abstraction.
- **ModuleSpecAdvertisement**: defines a module specification.
- **ModuleImplAdvertisement**: defines an implementation of a given module.
- **Rendezvous Advertisement**: describes a Peer that acts as a Rendezvous Peer.
- **Peer Info Advertisement**: describes the Peer Info resource. The primary use of this *advertisement* is to hold specific information about the current state of the Peer, such as uptime, inbound and outbound message count.

JXSEImplementor has been designed to take into consideration only three of the JXTA [54] *advertisements*: the *Peer Advertisement*, the *Rendezvous Advertisement* and the *Pipe Advertisement*. The first two are used by JIVES to define Peer resources inside the network of a JIVES Virtual Environment application, while the last one is needed to properly define the pipe communication channel used by the running application. An example of the Pipe Advertisement used by the JIVES applications is shown in Code Snippet 5.2.

Code Snippet 5.2: JIVES Pipe Advertisement

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jxta:PipeAdvertisement>
<jxta:PipeAdvertisement xml:space="default" xmlns:jxta="http://jxta.org">
        <Id>
                urn:jxta:uuid-59616261646162614E504720503250334A69766573424173A96384656D6F04
        </Id>
        <Type>
                JxtaPropagate
        </Type>
        <Name>
                JXTA: Jives Advertisement
        </Name>
        <Desc>
                Pipe Advertisement of JivesBasicDemo network
        </Desc>
</jxta:PipeAdvertisement>
```

The management of the network connection in the JIVES framework is totally transparent to the user: the JXTA [54] network starts automatically when the JIVES Virtual Environment application has been executed and stops when the JIVES Virtual Environment application exits. JIVES provides also the useful command *reset()*, used to restart the JXTA [54] network without exiting the application.

5.1.5 JiveScript grammars

SIMs and inline instantiation

One of the design objectives was making JIVES entities interoperable both by the system (fixed ways) and by the application (arbitrarily), and it appeared clear that this issue had to be resolved using JiveScript. It is necessary to allow the application to define specific behaviours and "mount" those into the management of the entities the system does; this need arises as soon as there is an application specific request that the system has to cope with, from rendering to producing the effects of an activation, from building application logic to defining those same entities. The idea behind the solution is achieved by inline instantiation of Java [51] interfaces using JavaScript, as Code Snippet 5.3 shows.

Code Snippet 5.3: Inline Instantiation

```
activable.bindAction(
    new org.jives.actions.ActivateNodeWithItemAction("useItemOnActivable",
        inventory, activable,
        new org.jives.sim.JivesActionModelIntf({
            execute: function (jivesEvent) {
                // Do something...
            },
            getDescription: function () {
                 return "Use item on activable";
            },
            render: function (jivesRenderableIntf) {
                 // Render the action...
             }
        })
    )
);
```

Regardless of the actual code meaning, in the above code it's possible to see that the application assign a new action to an activable entity. This action is partially implemented by default in the system, because activating an entity using an item is something you would expect to find in a Virtual Environment definition language. But then, there is something that the system doesn't know, the content of the *execute()* function that actually performs the activation job; which is very clear, instead, to the writer of the application logic. Functions like this one are methods of inline instanced interfaces called Structured Interactivity Models (SIMs).

Commands

As mentioned before, JiveScript is scripted Java [51]. Which in turn means that it has the complete grammar and semantic of JavaScript, extended to instantiate and access Java [51] classes. As an addition, it defines a few, simple commands to help the development of JiveScript application; anyone could extend this set of commands simply by inheriting the *JiveScriptEngine* class and creating new static methods in its subclass marking them with the *@JiveScriptCommand* annotation. Those methods will be parsed when the engine context is created and will be put available in JiveScript global scope as commands. The following list presents the commands introduced until JiveScript v 0.2:

- **name()**: display the current script name, set by the *__name()* directive.
- **load()**: load a JiveScript application, start the network and register this peer in the rendezvous directory.
  1. Parameter **path**: the path to the local resource.
- **me()**: returns a reference to the representation of self as a *JivesActiveNode*.
- **reset()**: reverts the system to the initial state, cleaning up memory.
- **echo()**: prints the textual representation of any object.
  1. Parameter **object**: the object to print.
  2. Parameter **keepLine**: (Optional) true to avoid line break. Defaults to false.
- **eval()**: evaluates a JiveScript. The result of the evaluation are put in the current scope, so writing "a = 2", for instance, equals writing "eval('a = 2')".
  1. Parameter **script**: the script to evaluate.

- **memstats()**: prints the current state of memory usage and availability.
- **entities()**: outputs the list of the contents of the JIVES registry.
- **version()**: displays JiveScript version.
- **implode()**: returns a JavaScript array of the Java objects passed as parameters. This is useful because there is no direct cast from a Java array to a JavaScript array.
  1. Parameter **params**: (Vararg) the Java objects to implode in an array.
- **saveState()**: persists the current state of the JIVES registry. The file will be saved as [name of the script]-[state name], where state name is the command parameter.
  1. Parameter **name**: the state name.
- **loadState()**: loads a saved state of the JIVES registry. The file loaded will be the one named after the same convention of *saveState()* command.
  1. Parameter **name**: the state name.
- **makeBag()**: given a Bag definition, returns a new *JivesBag* object.
  1. Parameter **definition**: the definition of the Bag, goes as follows: it is a JavaScript Object Notation (JSON) with one property, the bag ID. This property value must be an array, the collection of the Bag items.

Code Snippet 5.4: Bag definition

```
{
        bagId : [
                {
                        id : "",
                        categories : [
                                ...
                        ]
                        combines: [
                                // Some combination examples
                                [ new BagGraphHotspotIntf(),
                                null ],
                                [ null, "bagItemId" ],
                                [ new BagGraphHotspotIntf(),
                                "bagItemId" ]
                        ],
                        model: new BagItemModelIntf()
                        commonActionsRenderer: ...
                },
                {
                        ...
                }
        ]
}
```

Each Bag item is defined in JSON, too. It defines the following properties:

(a) **id**: (Optional) The ID of the Bag item, useful to refer to it, will be generated if omitted.

(b) **categories**: an array of strings that defines the categories this item is into. The item will be in any case categorized under CATEGORY_DEFAULT, even if this is empty.

(c) **combines**: an array of combination definitions. Each combination is an array with two fields; the first is a *BagGraphHotspotIntf*, the second is an ID of another Bag item. In the bag graph will be bound a combination between two nodes (the object being defined and the one whose ID is specified in the combination), using the given HotSpot as an arc. Either of the combination fields can be null, as seen in Code Snippet 5.4. If the ID field is null the arc is dangling, while if the HotSpot field is null, the combination happens without need to specify any HotSpot (simply selecting the right parent items).

(d) **model**: a *BagItemModelIntf* that will be used as model when this item is activated or represented.

(e) **commonActionsRenderer**: if this property is set, a set of common actions will be automatically bound to this item. The available actions will be:

   i. **Select/Deselect**: allows to extract a certain quantity of this item from the Bag.

   ii. **Examine**: allows to display a description for this item.

   Note that the value of this property must be a *JivesRendererIntf*.

2. Parameter **model**: a *BagModelIntf* that will be used as model when this inventory is activated or represented.

3. Parameter **combiner**: a *BagItemCombinerModelIntf* that will be used as model when the end user tries to combine items from this inventory.

- **makeDialogue()**: given a dialogue definition, return a new *JivesDialogue* object.

1. Parameter **definition**: the definition of the dialogue, in JSON, is again a single property with value, where the property is the ID of the dialogue and its value a collection of dialogue requests.

Code Snippet 5.5: Dialogue definition

```
{
        dialogueId : [
                {
                        id : "",
                        className : "",
                        question : "",
                        answer : "",
                        actions : [
                                // Actions examples
                                myActionObject,
                                new org.jives.actions
                                        .JivesActionIntf( ... )
                        ]
                },
                {
                        ...
                }
        ]
}
```

Each dialogue request is defined in JSON, too. Advantage of this design is the possibility of reusing objects in the definitions or even instancing entities inline, as it can be seen in the actions definition of Code Snippet 5.5. The properties defined in each dialogue request are:

(a) **id**: (Optional) The ID of the dialogue request, useful to refer to it, will be generated if omitted.

(b) **className**: (Optional) class name of the request instance, one of *DialogueRequestIntf*. Defaults to *org.jives.dialogues.DialogueRequest*.

(c) **question**: a string for the dialogue initiator to present as question.

(d) **answer**: a string to receive as corresponding answer.

(e) **actions**: performed when the request being defined is issued. If the dialogue should go on, it must explicitly define a *StartDialogue-Action* here.

2. Parameter **model**: a *DialogueModelIntf* that will be used as model when this dialogue is activated.
3. Parameter **renderer**: a *JivesRendererIntf* that will be used for the dialogue request representation.

Directives

Beside of commands, there are three important directives that compose the Jive-Script syntax in version 0.2; they must be used to instruct the interpreter about the environmental conditions into which the script runs.

- **__name()**: sets a name for the current script. There can be no unnamed scripts in fact when the interpreter starts or it is reset, a new name for the current script is automatically generated, but this directive is suitable to assign an human readable name. Setting a name is important because it will be used to register in the Rendezvous directory both the Peer and the application that it is running.

  1. Parameter **name**: the script name.

- **__uses()**: declares a dependency upon a certain Implementor, or action, or generally a Java [51] class or package. There can be multiple calls to this directive, one for each dependency that a script expects to find when it is run. When a script is loaded, the interpreter first checks that all the dependencies are satisfied and then proceeds to its execution.

  1. Parameter **dependency**: the fully qualified name of a dependency.

- **__scripting()**: sets the flag to allow/disallow scripting from the moment it is parsed, on. Once is set to false, only *reset()* will enable scripting again. As "scripting" we considered any of the following operations:

  Instance or *make()* any JIVES objects.

  Execute multi-line commands.

  Execute variable assignments.

  1. Parameter **scripting**: a boolean flag telling to enable or disable scripting.

5.1.6 Critical conditions and their resolution

This section briefly describes many situations that put the design under stress and how those issued were managed or resolved, confirming the validity of the work done so far.

TradeItemsAction: Peer disconnection

When a Peer disconnects during a trade action, it can return to a persistent state right before the transaction happened on its reconnection, if the application wishes this to happen. So, there's no problem of data loss on the disconnecting part, but the counterpart should revert the transaction locally. Figure 5.5 is a sequence diagram that describes this automatic procedure.

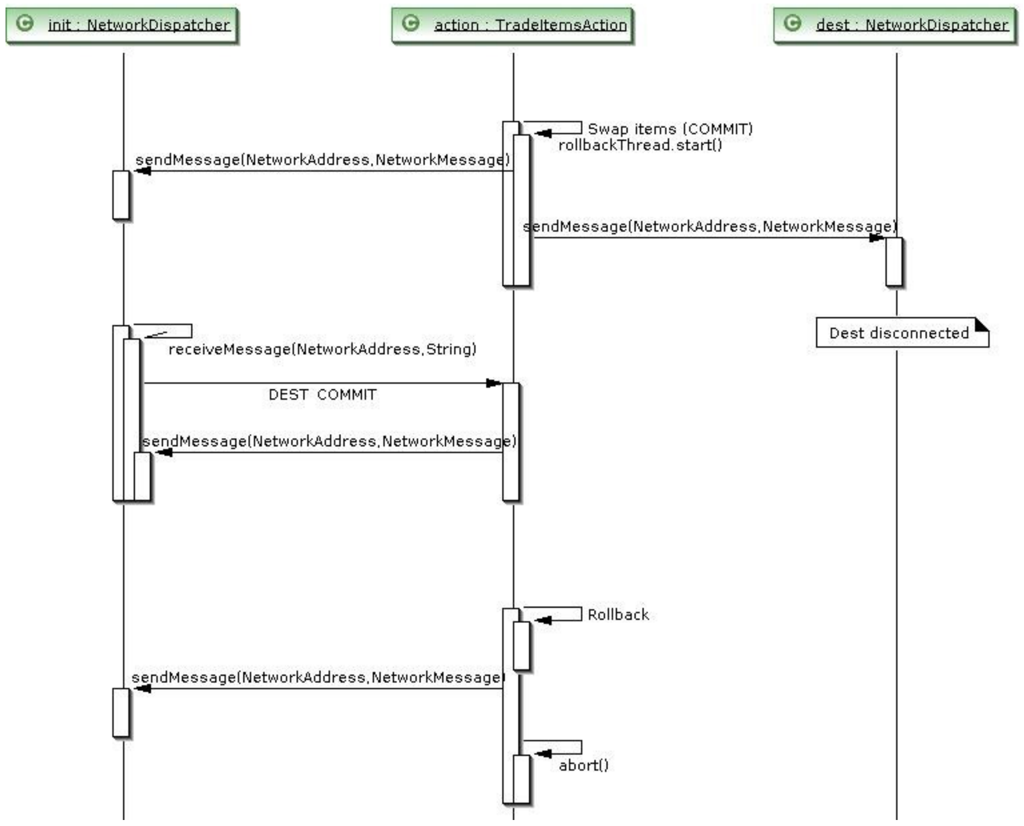

init : NetworkDispatcher
action : TradeItemsAction
dest : NetworkDispatcher
Swap items (COMMIT)
rollbackThread.start()
sendMessage(NetworkAddress,NetworkMessage)
sendMessage(NetworkAddress,NetworkMessage)
Dest disconnected
receiveMessage(NetworkAddress,String)
DEST COMMIT
sendMessage(NetworkAddress,NetworkMessage)
Rollback
sendMessage(NetworkAddress,NetworkMessage)
abort()

Figure 5.5: TradeItemsAction rollback

Initiator part might or might not have received destination commit acknowledgement (DEST_COMMIT) before destination disconnected; what matters is that destination will not send back the INIT_COMMIT acknowledgement and this means that Initiator cannot go into INIT_AND_DEST_COMMITTED state before rollback countdown reaches zero, so the transaction is aborted.

Persistence of complex classes

All JIVES core classes are heavily dependent on the other parts of the framework; even more important, not all their fields are serializable, thus it's simply not possible to store them as data, being that an XML representation, a serialization or a database entry.

To overcome this limitations, *JivesObjectIntf* exposes a *freeze()* method to save the entity and an *unfreeze()* method to restore it. What happens in this methods is that an apposite storage (implementing *PersistenceIntf*) is used to store (on freeze, then restore on unfreeze) the state-critical fields of the entity instead of storing the whole entity. This approach allows to use the *setProperties()* method of *JivesRenderableIntf* to produce right results in even more specific cases. For example, let's consider the case that ”self” active node is restored; that one is actually a *RemoteActiveNode*, that has a network address. The persisted network address, though, is no longer consistent and only a fine-grained filter allows the old state to be appropriately merged with the new state.

Concurrency and thread safety issues

A JIVES application cannot be restricted to a single thread design. It's indeed convenient that Implementors rely on their type-specific threads (like the render thread in a Presentation Implementor, or the network messages dispatcher in a Network Implementor) but also JIVES events might be fired forking the program execution: all the JIVES entities involved in this concurrent situation must be kept consistent from one thread to another. In the Java [51] language, each thread is awarded his own stack, but all the thread share the same heap. In other words, changes applied during runtime to the values of statical allocations will not be reflected from one thread to another, while in the case of dynamic allocations they will. The Java [51] keyword *synchronized* is used to avoid race conditions on a block of code, meaning that only one thread will be executing that code from the beginning to the end, even if the operating system put the currently running thread to sleep and activate another thread that is queued for that same code execution.

Many methods of the *Jives* class are synchronized: *generateId()* so that there will always be a locally unique ID. *loadState()* and *saveState()* to avoid concurrent changes in the entities when they are persisted or restored. The *doLoop()* internal behaviour, which consists of network updates and processing of events; the latter case is especially important because event actions can actually concurrently execute other events. It happens that those actions are executed in separate threads but want to manage entities reflecting changes of an action to another; in this case the only way to achieve the right result is checking out a copy of those entities from the JIVES registry and then check back in when the appropriate changes have been performed. This is why often code does not refer to *this* when accessing even the protected fields of a class, but uses the registry instead, or an instance of the same class passed in as a parameter, like it happens in *TransactionIntf.nextState()* method.

jME HotSpot Combinations helper

A combination that involves HotSpots can be very straightforward to think, because the user can visually identify the possible candidates. However, it can be extremely difficult to realize: a 2D representation of a 3D space doesn't give the perception of depth and this is a major issue when it comes to aligning precisely two points in space. To help the user in this situation, the Implementor provides an automation that takes care of the orientation and position of the chosen

HotSpots so that once they have been selected by a mouse click they become immediately aligned and at that point it's sufficient to get the two items close enough to perform the combination (by means of the middle mouse button or the space bar). The algorithm cannot perform the alignment in terms of a pure rotation, because there are too many degrees of freedom to affirm if the rotation involves one, two or perhaps all of the three axes. Consequently, the best approach involves one rotation and one translation.

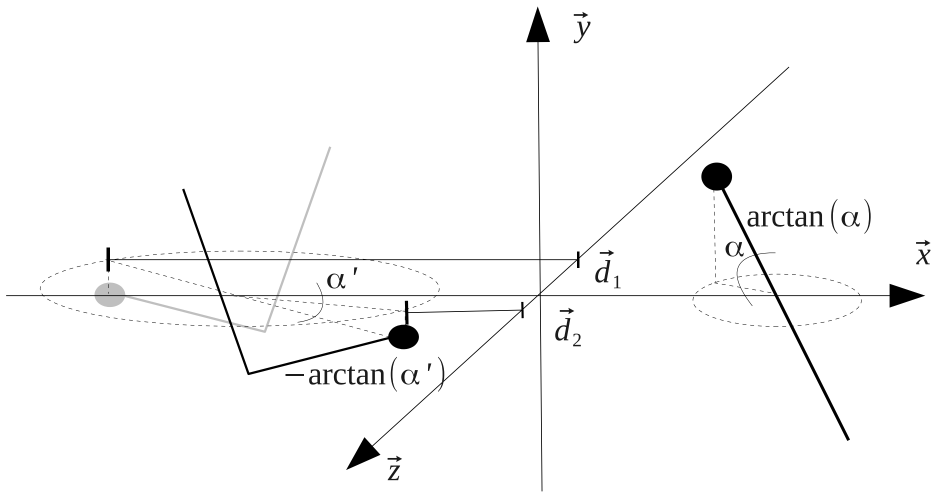


Figure 5.6: Rotation step of the combination helper algorithm

The rotation, represented in Figure 5.6, reaches the objective of putting the HotSpots on the $\vec{x}\,\vec{z}$ plane (jME [31] reference system is right handed) by having the chosen HotSpots face each other toward the origin, which is set to be the median point of the two objects pivot on the $\vec{x}$ axis. In order to accomplish this, it infers the angle of rotation from the y-coordinate of the HotSpot in its local (object) space, because this value is the arctangent of the unknown angle; in the case that the coordinate is negative it takes the opposite of the result. There is uncertainty whenever the rotation brings the HotSpot to face outward from the centre: to resolve to the right placement, the rotation is performed two times taking in account the resulting z-coordinate of the HotSpot: the second time a rotation angle of π is applied to the $\vec{y}$ axis. The right rotation is the one that minimizes the distance from the centre on the $\overrightarrow{z}$ axis: in Figure 5.6, $\overrightarrow{d_2}$ is preferred to $\overrightarrow{d_1}$. At this point, the situation is that one depicted in Figure 5.7, where there might be an offset on either or both the $\overrightarrow{y}$ and the $\overrightarrow{z}$ axis. A compensating translation ($[0, \overrightarrow{o_y}, \overrightarrow{o_z}]$) is applied, so that the two HotSpots lie exactly on $\overrightarrow{x}$.

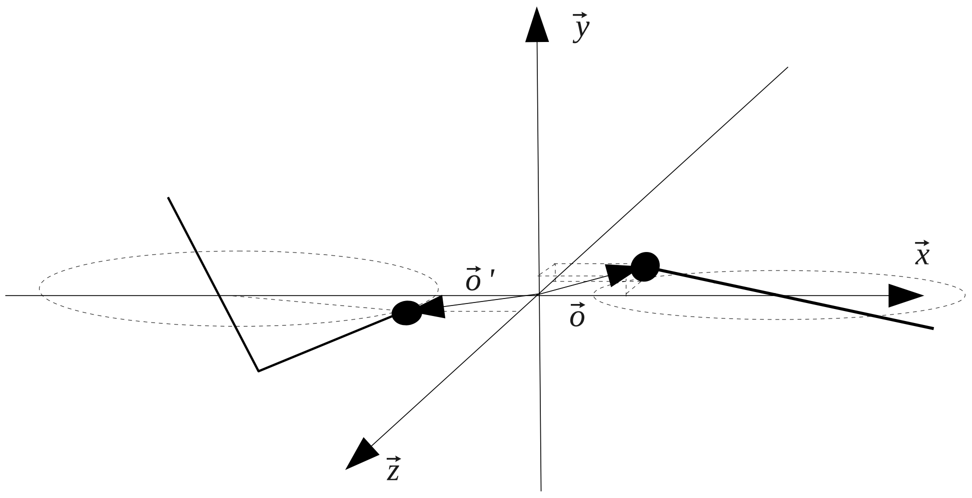


Figure 5.7: Translation step of the combination helper algorithm

Overcoming JXSE v2.6 single-Peer limitation

By design, prior to v2.7, there was no possibility of restarting the network without restarting the virtual machine. *JXSEImplementor* uses JXSE 2.6 [54], the stable version at the time of its writing, so it had a huge problem in dealing with JIVES reset and restart dynamics because only a single instance of a World Peer Group could be instantiated at a single time in the same JVM. The solution was hiding each peer behind its own multi-instance endpoint class-loader so that in the same virtual machine they all had the opportunity to make a fresh start. Of course, "dead" Peers must be subject to de-initialization and garbage collection appropriately, while running Peers cannot have their method invoked directly because they reside in different class-loaders, but the multi-instance endpoint deals with the issue using Java Reflection API to communicate. The class-loader implements a static cache so all the classes are loaded once, the first time a peer is initialized;

then every time a new peer starts they are all immediately available from memory.

5.2 Implementation

The implementation of the JIVES framework is accurately described in the following sections, by following the order dictated by the VESLOM Design. Firstly, the Universal Platform and all its components, secondly the Networking layer, thirdly the Middleware layer and finally the Application layer are explained and analysed in their implementation.

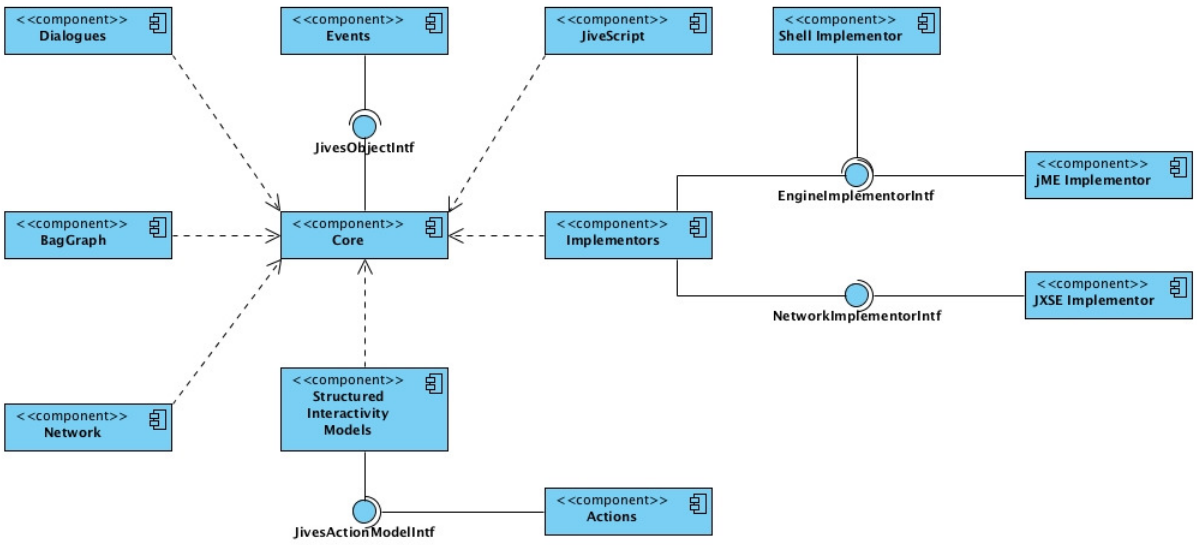


Figure 5.8: JIVES Component Diagram

The Component Diagram shown in Figure 5.8 depicts at a high level the structure of the framework.

5.2.1 Universal Platform

The JIVES Universal Platform includes the core classes and all the basic functionalities of the framework. In particular, the Universal Platform is composed by the JIVES Core, which represents the kernel of the framework; the Structured

Interactivity Models (SIMs), needed to define specific behaviours for a Virtual Environment; the Persistence Layer, responsible to load and save the game state of the application; the Actions and Events Management System, in charge to manage all the actions and events that can occur in the Virtual Environment application; the Dialogues Manager, which lets the developer instantiate dialogues between Playing Characters (PC) and between Playing Characters and Non-Playing Characters (NPC); the Bag Graph, related to the management of the objects available in the Virtual Environment and their relations.

JIVES core

The Core classes represent those identified to be basic entities of the environment, namely Actions, Active Nodes, Bags (inventories), Bag Items, Dialogues, Scenes, Events and Event Listeners. All of those implement the *JivesObjectIntf* interface that exposes an unique identifier string. A working copy of a JIVES object can be checked in or out from the JIVES registry using its identifier. The purpose of using the registry is that of creating logical references instead of pointers to memory (which would incur in cyclic references) and the ability to retrieve an object that is needed at any time anywhere in the code without having to propagate widespread and intricate references.

Furthermore, being JIVES by nature a distributed system, it arises the need to store user data from the volatile memory into a persistent secure location for each and every peer. This way, it is possible to restore a consistent state later on. Accordingly, the objects in the JIVES registry represent the state of the application. The security is a crucial aspect to bare in mind due to the fact that external modifications of the persistent location would lead to an incoherent situation or perhaps would advantage one part at the expenses of the others. Of course, a server side double check of the user data integrity can be introduced on top of the minimal but efficient mechanism that the universal platform guarantees, which consist in an encrypted, local, object database.

All of the JIVES object share a common pattern: an internal management that the system deals with to put the rails on the object behaviour and the external implementation, which resolves to the methods the implementor defines or the programmer code injected by scripting. This dual nature allows JIVES to take care of the expected features (for instance, action execution, inventory management, traversing scenes and so on) while it allows to freely define specific features that typically the application must cope with (for example the effect of an action being executed, the shapes and quantities of the items in the inventory, the way an end user traverse from one scene to another). This duality is achieved in Java [51] without the usage of multiple inheritance but with delegation instead. This is why the above described model resolves to a list of interfaces, the Structured Interactivity Models (SIMs), whose implementation, changing from one application to another, gives shape exactly to the needed virtual environment.

A last issue comes with the representation of the JIVES objects. Being this a tipical implementor issue, it differs depending on its capabilities, meaning that a command line interface would have a textual representation of the objects while for instance a graphic engine would render 3D geometry to represent them. To decouple system rendering calls from their real implementation, a set of renderable entities is defined and to each of them a respective renderer from the implementor is associated, in order to maintain a very high degree of interoperability.

Structured Interactivity Models

The interfaces contained in package *org.jives.sim* are used to create JIVES objects in JiveScript or in Java [51] and are called, in short, models. Implementing those interfaces, the programmer defines the appropriate behaviour for his virtual environment. Both Java [51] and the scripting API *javax.script* [52] JavaScript support inline instantiation of interfaces and using this approach it becomes possible to inject programmer's code into the system architecture. For example, models of any Activable present the *getActivationEvent()* virtual method, where it's possible to specify which kind of event will by fired by the activation, being this a crucial step for the application logic.

One could write the whole models in JiveScript but this gets rapidly infeasible as the complexity of the application raises. The typical approach is to encapsulate the most of the application features into the implementors and rely on the methods they expose when writing the application logic in JiveScript. By doing so, the same, good-written, general enough implementors can be use to create a huge amount of different application scripts.

Persistence Layer

JIVES allows to save and load state snapshots of its register. Under the hood, every registered object extracts its field data into a proxy object (one of the package *org.jives.core.persistence*); not the whole object is extracted, due to serialization and redundancy issues, but a subset of the fields that characterize its state. Then the proxy object is saved in an object database. The way back, during state restore the proxy object is read from the database and the respective registry entry is updated.

The advantage of this approach is the decentralization of the distributed application state. Of course, the main issue is that of avoiding local and extraneous modifications to the database so that the integrity of this distributed state is kept. The most obvious solution is that of encrypting the database and to reach this objective the system must obtain a secret key that cannot be generated locally and that the peer must not be aware of. To achieve this, the Network Implementor must contact a remote location, identify the peer with his credentials (account and password) and return a secret key to be used for encryption. As security measure this connection must happen in HTTPS, so that it is possible to avoid man-in-the-middle attacks.

Actions and Events Management System

JIVES performs event processing and other network related operations every time the *doLoop()* method is called in the application main loop. Actions and events are tightly coupled, because every time an activable entity is activated by mean of an action, an activation event is fired. But events can also by fired independently from entity activation, at any time, to tell to the program logic that something has happened and consequently the action bound to the fired event must be executed. Event listeners can be registered in order to intercept any event and decide which reaction to undertake, even inhibiting the execution of the action bound to the intercepted event is an option. It's important to notice that firing an event is not always the same thing as saying that the event happened, because every fired event can be kept in a pool while they wait to be processed, but only when the logic of an event *isHappened()* method is satisfied the action bound to it is executed.

There are different types of events, namely:

- **Generic events**: the default ones, those introduced by the program logic to perform its own tasks.
- **Activation events**: occur every time an activable entity is activated by mean of an action.
- **Remote receive events**: used by the system to synchronize the distributed state, occurs when the local peer receives an update from the network.

Any event can be repeated many times because it will stay in the pool of fired events as long as it's needed, depending on its retention policy. The possible values of this policy are:

- **Discard always**: The event is discarded as soon as it is retrieved from the pool, independently of the fact that it has actually happened or it is likely to happen in the future. This is useful when it's necessary to pre-emptively fire an event and it is only possible to tell in a second moment if the action bound to that event must be executed.

- **Discard when happened**: The event stays in the event pool as long as it's not happened, then its action gets executed and the event is discarded.
- **Keep always**: The event will be processed and possibly its action will be executed as long as its execution policy doesn't change.

The event processing usually happens immediately when the event is fired but, as mentioned above, this is not always the case because this behaviour can be influenced. Moreover, it's not always desirable to wait for the event process to complete before the main thread execution continues. Those cases are regulated by the following flags:

- **Execute As Soon As Possible (ASAP)**: The fired event will stay in the events pool until JIVES loop starts, and it will be processed in that loop.
- **Fork and execute**: The fired event will execute its action in another thread, so that the main thread will not be kept waiting. This comes handy while waiting the user input - which is a blocking operation - in some calls to the SIMs.

When it comes to actions, JIVES offers a minimum set of common actions and a hierarchy of Activable. Objects like Active Nodes, Bags, Bag Items, extends this class that allows actions to be performed on them and that keeps trace of their kinship, like the case of activable items in an activable inventory. One could possibly extends himself this hierarchy to define other types of activable entities that have not been covered so far.

Common actions are those expected features like activation of an Active Node using a BagItem, adding and removing items from the inventory, traversing portals, starting dialogues and trading items. The *TradeItemsAction* is not the same as the other actions because exchanging objects between different peers on the network implies the concept of transaction, which is supported at a very low level because of the high abstraction (every transaction simply implements the *TransactionIntf* interface, which is just a guideline), however:

- **Atomicity** can be guaranteed by a timed out roll-back if commit is not confirmed by both parties.
- **Consistency** is assured for every transaction: the distributed state of the application remains consistent as long that each and every Bag Item traded from one peer to another is respectively given from the former and received by the latter.
- **Isolation** is enforced by mean of an EventListener to prevent other JIVES Actions to occur on the requested activable while the transaction is going on.
- **Durability** is guaranteed in the moment that the registry is persisted, which is a good idea after an important transaction occurred.

Dialogues Manager

JIVES dialogues can be divided in two categories. Playing Characters (PC) dialogues, which are chat-like interactions between the avatars of human users, and Non-Playing Characters (NPC) dialogues, that allows an human user to acquire informations from the environment in a multiple choice, interactive, question/answer paradigm.

In both cases the entrance into dialogue mode consists in the activation of an Active Node using a *StartDialogueAction*. While PC dialogues are simple chat and don't offer any other feature than textual communication between two or more peers in a chat room, NPC dialogues are much more versatile. In the latter, there can be multiple paths of interrogation that lead to different results, but that in general do nothing more than triggering actions - any kind of action. So, for instance, its possible to acquire inventory items by talking to NPCs, or playing cinematics and whatever else.

Dialogues are defined in JiveScript with a very rigid JSON (JavaScript Object Notation) that define each Dialogue Request in a list; every request specifies a question and its answer, plus the actions triggered by choosing that Dialogue Request. To create a dialogue, the JSON object is passed as parameter of the *makeDialogue()* JiveScript command. The dialogue can be attached to any Active Node by binding a new *StartDialogueAction* on it.

Bag Graph

Underneath a JIVES Bag object, the delegation resolves to the Bag Graph and to a SIM. This graph defines which items can be contained in this inventory, which are their possible combinations in order to produce more Bag Items and how this combinations must be performed interactively by the end user; on the other side, the model exposes methods that need to be implemented so that it's possible to activate the Bag and the items contained in it.

The graph is first built by defining each node of the combinations tree, then it gets queried when trying to combine two items, represented by two respective nodes in the graph. The final shape of the Bag Graph is a forest of directed graphs in which each node (that represent a combination of two parent items) has either zero or two incoming arcs (the parent items). If the node has no parents, it is defined to be a root, while every child which is not parent for any other combination is said to be a leaf. A *BagGraphHotspotIntf* is associated to each arc of the graph; this information is useful during interactive combination to give a more realistic scenario in which not only two coherent items must be selected to be combined, but they can present many hotspots that match a combination result.

Moreover, the nodes of the graph are tagged with one or more labels that subdivides Bag Items in different categories, to improve searching and indexing operations. At any time new nodes can be added to the graph, as well as bound nodes can be removed in cascade, to fully support the dynamic changes in the inventory structure.

Just like dialogues, Bag Graph is defined in JiveScript using JSON. For every Bag, one defines a list of Bag Items specifying their id, the categories they belong to, every possible hotspot and the id of the respective child and the model of the item. Then, using the *makeBag()* JiveScript command, the inventory is created empty, with the appropriate underlying Bag Graph.

5.2.2 Networking Layer

The Networking Layer has the responsibility to retrieve resources across the network, and to manage all the functionalities pertaining networking operations. The Networking Layer implementation provided by default in the JIVES framework is completely based on the JXTA [54] Peer-To-Peer technology.

JXSE

The JIVES Networking Layer is implemented using the Open Source Peer-To-Peer JXTA [54] technology, in particular its Java [51] version, JXSE. Being JXTA a Peer-To-Peer technology, its functioning is based on the distinction between Rendezvous, Relay and Peer. A Rendezvous is a super peer with the responsibility to assist normal peers and handle message broadcasting; a Relay is a peer used to store and forward messages between peers that do not have direct connectivity because of firewalls or NAT; a Peer participates in the Peer-To-Peer network implementing only the basic JXTA functionalities. The JIVES implementation provides a breakdown of Peers into two main categories: normal Peers and Rendezvous Peers with Relay capabilities. In that way, in order to have a fully working network, it is sufficient that there are at least two peers, one of which is a Rendezvous/Relay peer. The JXSE Implementor has been designed to fully working both in a LAN network and a Internet network. The JIVES user has the possibility to choose this modality by correctly set up the JIVES Network Settings. Both in the LAN modality and the INTERNET modality, the network has to be started by a Rendezvous/Relay. In the LAN modality, Peers who want to connect to the network have to know the external IP of the Rendezvous/Relay, and insert it into the JIVES network settings. Differently, in the Internet modality, Peers do not have the possibility to know in advance the external IP of the Rendezvous/Relay connected at the time. In order to overcome this issue, JIVES provides a mechanism whereby any Rendezvous/relay running a JIVES application is registered in a Rendezvous directory hosted by a dedicated server. A Rendezvous directory sample is fully reported in Appendix D. The default server adopted by JIVES is provided by `sourceforge.net` [24], but the developer is encouraged to use any other hosting server, just specifying the URL in the JIVES network settings. Associated to each Rendezvous/Relay registered on the server there is a Token, composed by the Peer ID of the Rendezvous/Relay and the name of the JIVES application currently running. Through the use of the Token JIVES ensures that only the legitimate Rendezvous/Relay is able to perform writing operations on the Rendezvous directory, such as updating the information status or unsubscribing from the list. Obviously the writing operations on the Rendezvous directory possible by a Rendezvous/Relay are related only to the Rendezvous/Relay itself, because it is not possible to modify the informations of an other Rendezvous without knowing its own Token. In order to manage a sudden disconnection of a Rendezvous/Relay, the JIVES implementation provides that the server monitors continuously the existence of the Rendezvous registered on the Rendezvous directory. In particular, each Rendezvous/Relay sends to the server once per minute an update message: if the server does not receive any communication from a given Rendezvous/Relay for more than a minute and a half, the server considers disconnected the Rendezvous and unsubscribes it from the Rendezvous directory.

Any Rendezvous/Relay, while registering on the Rendezvous directory, transmits also the MD5 of the whole JIVES script currently running. This is an additional layer of security which permits to be sure that only the peers running the same JIVES application (with the same MD5) are interconnected in the network related to the application itself. When a user starts a JIVES application the Internet modality, JIVES checks if the there are already Rendezvous running the same application: if so the user is connected to the network as a normal peer, if not JIVES automatically makes the user a Rendezvous/Relay and registers it on the Rendezvous directory.

Code Snippet 5.6: Rendezvous/Relay run method

```
while (running) {
        // Adding the Rendezvous/Relay as a Discovery listener
        discovery.addDiscoveryListener(this);
        Tools.sleep(1000);

        // Publishing the peer advertisement
        JXSETools.publishPeerAdvertisement(netPeerGroup, discovery);
        Tools.sleep(1000);

        // Retrieving remote Advertisements (looking for peers)
        JXSETools.retrieveRemoteAdvertisements(discovery, this, null);
        Tools.sleep(1000);

        if (!XMLConfigParser.getLanChoice()
                && XMLConfigParser.getInternetChoice()) {
                // Updating time only when we have a Internet connection
                FileManager.getInstance().updateTime(IPv4, IPv6, tcpPort,
                        RendezvousRelayManager.getRendezvousID(),
                        RendezvousRelayManager.getScriptName(),
                        RendezvousRelayManager.getScriptMD5());
        }
        // Clear the interrupted status
        Thread.interrupted();
        Thread.sleep(27000);
}
```

The JXTA [54] technology is a Peer-To-Peer technology based on the publishing of Advertisements. Advertisements are language-neutral meta structures represented as XML documents and are used to describe and publish the existence of a Peer's resources. JIVES implements three types of Advertisements: the Peer

Advertisement, the Rendezvous Advertisement and the Pipe Advertisement. The Peer Advertisement is used to hold specific information about a normal Peer, the Rendezvous Advertisement describes a Peer that acts as a Rendezvous/Relay and the Pipe Advertisement describes the pipe communication channel used by the JIVES Peers to exchange messages. Each Peer, both a normal Peer and a Rendezvous/Relay, cycle constantly publishing its own Advertisement and looking for Advertisements of other peers, as shown in Code Snippet 5.6.

Messages are sent through a pipe communication channel. This channel brings messages without any kind of filtering operation: when a Peer sends a message, all the Peers listening the given communication channel receive that message. Obviously, the developer can choose to use this kind of configuration, or implement a specific Interest Management. Interest Management is the term commonly used to describe restricted message dissemination between objects or avatars using Virtual Environment division. According to Morgan [39], Interest Management is classified into two main categories:

- ***Region-based Approach***: the Virtual Environment is divided into well defined regions that are static in nature. The recipient of a message is limited to only interested participants within the same or neighbouring region as the sender. When an object traverses a region boundary region membership must be updated. JIVES is particularly predisposed to implement this kind of approach: when developing a JIVES Inventory-based Virtual Environment application, the developer, by creating a new Networking Layer Implementor, has the possibility to implement a division of the whole logic of application according to a specific number of scenes or rooms, which can represent the regions of the Region-based approach. In that kind of configuration, only peers participating the same room can message each other.

  A variant of the Region-based approach came to the attention within the last years with the growth of satellite technologies: the management of the

Peers of a Virtual Environment may be implemented through a geolocation mechanism, in order to allow a specific Peer to interact with others Peers only if they are geographically close to him, improving in that way the quality of the online experience, or to allow a user to perform a certain action in the Virtual Environment only if he is physically at a given place on earth. To achieve this result, it would be possible to exploit the Rendezvous directory for the logical separation of Peers when they connect.

- ***Area-Of-Interest Approach***: each object or avatar is associated with an Area-Of-Interest or Aura that defines an area of the Virtual Environment over which an object or avatar may exert influence. An Area-Of-Interest or Aura may be simply modelled as a sphere that shares its centre with the positional vector of the object or avatar it is associated with. Nevertheless this kind of approach is more difficult to implement with respect to the Region-based approach, the developer may add this functionality to the JIVES framework by developing a new Networking Layer Implementor that encloses this feature.

A third approach may be implemented in the JIVES framework: according to the JXTA [54] Network Architecture, the Peers participating a Virtual Environment can be divided into several Peer-Groups, each of them with a specific Peer-Group ID. In that way, the developer, after implementing a new Networking Layer Implementor with this particular feature, can manage the dissemination of messages according to the subdivision into Peer-Groups: a message may be received only by some Peer-Groups and not by another ones. This kind of approach may be called *Group-based Approach*. JIVES has been designed to implement a subdivision of Peers according to the Virtual Environment application they are running: only Peers executing the same JIVES application can communicate each other. Within the application JIVES does not implement any kind of Interest Management: each message is sent fully broadcast to every Peer participating the same JIVES application. The possibility to implement an Interest Management scheme is left to the developer.

JIVES Peers communicate using JXTA [54] messages: each message is sent through the Transmission Control Protocol (TCP) in order to have a reliable transmission. In the JIVES implementation each message consists of a serialized object, called *NetworkMessage*, in which there are the basic informations needed for a given action or transaction.

Code Snippet 5.7: NetworkMessage constructor

```
/**
 * The constructor
 *
 * @param senderAddress
 *                              the NetworkAddress of the sender
 *
 * @param type
 *                              the type of the message: it can be
*                              TYPE_REMOTE_DIALOGUE;
 *                              TYPE_REMOTE_ACTIVE_NODE_UPDATE;
*                              TYPE_REMOTE_ACTION
 *
 * @param sceneId
 *                              the ID of the current scene
 *
 * @param payload
 *                              the content (payload) of the message
 */
public NetworkMessage(NetworkAddress senderAddress, int type,
String sceneId, String payload) {

        this.senderAddress = senderAddress;
        this.type = type;
        this.sceneId = sceneId;
        this.payload = payload;
        this.forcedLocalDispatch = false;

}
```

As shown in Code Snippet 5.7, the *NetworkMessage* object includes informations about the sender of the message, the type of the request and the scene currently visited by the sender. JIVES provides different types of requests: the *TYPE_REMOTE_DIALOGUE* message allows a user to start a dialogue with an other Peer; the *TYPE_REMOTE_ACTIVE_NODE_UPDATE* message specifies that the given message is an update message sent by a Peer to all the Peers connected to the same network, in order to communicate some changes in the properties of the sender Peer, such as the position, the colour, the ID of the currently visited scene; the *TYPE_REMOTE_ACTION* message is used to perform actions between the Peers connected to the same network. In particular, when using a *TYPE_REMOTE_ACTION* message, the *payload* of the serialized object *NetworkMessage* includes the given Remote Action that has to be executed in the JIVES Virtual Environment, specifying the source and the target of the operation.

JIVES provides also an additional layer of management of the Peers connected to a JIVES application. It is possible to associate to each Peer an account with a given password, thereby obtaining a database of all the users who have logged at least once a given JIVES application. For each account the server is responsible to release a Secret Key, through which it is possible to encrypt the user data, giving in that way the possibility to the JIVES application user to restart the Virtual Environment later without losing the application state related to his avatar.

5.2.3 Middleware Layer

Apart from the network access, commonly a JIVES application will need at least to display JIVES entities and make end user interact with them; on top of those, a more complex application would develop any more sophisticated features it requires. This kind of concepts are extremely abstract and heavily rely on the container JIVES is inserted into. For the interoperability of JIVES to be as much generic as possible, there is no practical way to define a structure in the middleware for every implementors to follow in order to work.

As a matter of fact the Middleware Layer is left open to any type of implementation, to cope with radically different needs. It is easily understandable that this can bring to a wide variety of applications, that can only work with their respective implementor; this means the same script should be ported to different implementors, because it simply won't work with all possible implementors out of the box. However, to demonstrate the validity of the generic and versatile design over the rigid, full portable, statical presentation structure, two very different implementors are proposed and discussed further on, and the same JiveScript application was ported to both of them.

Shell Implementor

This implementation is a MUD Object Oriented (MOOs) that resembles those of early eighties adventure computer games in which the player would read descriptions of the situation his avatar was into and would react by typing commands in a shell to move around, interact, operate the virtual environment and the inventory and so on.

A list of instructions can be stored as a program and then executed. The command line interface is equipped with code completion and suggestions to ease the rapid writing. Although a complete list of commands is available entering the *help()* command, three main commands allow the end user to interact with the virtual environment; this is a huge simplification of all the underlying complexity. Those commands are:

- ***lookAround()*** returns the list of surrounding entities.
- ***activate()*** perform activation of the activable whose id is passed as parameter.
- ***move()*** allows the avatar to move around in the virtual environment.

Of course one is allowed to write as many SIMs (Structured Interactivity Models) as needed directly in the script, however, for ease of use and reuse, the following built-in SIMs are made available to the JiveScript by this implementor:

- ***ActiveNodeRenderer***: instanced when generating individuals and artefacts.
- ***BagItemRenderer***: model of the bag items.

- ***BagRenderState***: model of the inventory.
- ***GUIState***: used to output notifications and to acquire user input.
- ***HotSpotRenderer***: model of the Bag Graph HotSpots.
- ***SceneRenderState***: model of the scene that allows to quickly setup remote dialogues and trade transactions for the avatars in the scene.
- ***ShopCatalog***: used to setup trade transactions with NPCs.

The Shell Implementor extends the JiveScript internal engine based on the scripting API *javax.script* [52] and introduces its own commands, but it is actually not much more than a textual user interface that delegates the work to its superclass. Once this implementor is introduced, every sort of command line, MUD, MOOs, adventure computer games or textual simulations are ready to be produced simply by writing the appropriate JiveScript, but its real purpose remains mostly that of development and debugging of new features for JIVES.

jME Implementor

Being the objectives of this thesis focused on 3D Virtual Environments, there was the need to code an Implementor that would allow the graphical representation of JIVES entities. The jMonkeyEngine [31] turned out to be a good product to easily integrate JIVES with.

There are two main issues to deal with when introducing this kind of implementation of the Presentation Layer. The first is the actual rendering of geometry, the second the need to display and operate a Graphical User Interface (GUI); jME [31] can resolve both issues within itself, offering a scene-graph approach to render geometry and an internal integration with Nifty GUI [43]. The work that needed to be done was translating the Shell Implementor models into their jME [31] equivalent, thus many of the available SIMs share at least the name. The complete list is the following:

- ***BagCombinerState***: state enabled when the user has to combine different items interactively if they define a combination that requires HotSpots. The items can be rotated using the left mouse button, translated with the right mouse button and an attempt to combine them is realized pressing the middle mouse button or the space bar. The combination item is obtained if the right HotSpots get close enough. For demonstration, HotSpots are represented by blue, transparent spheres but of course can be implicit.
- ***BagItemRenderer***: model of the bag items.
- ***BagRenderState***: model of the inventory.
- ***GUIState***: used to encapsulate Nifty [43] interface and common GUI related operations, like notifications, updates and so on.
- ***HotspotRenderer***: model of the Bag Graph hotspots.
- ***NPCRenderer***: used to introduce Artefacts in the scene.
- ***PlayingCharacterRenderer***: used to represent the avatar of "self" (the local player) and those of the other people running the same application.
- ***PortalRenderer***: used to represent doors that allow access to different scenes defined in the application script.
- ***SceneRenderState***: model of the scene that allows to quickly setup remote dialogues and trade transactions for the avatars in the scene.
- ***ShopCatalog***: used to setup trade transactions with NPCs.

This isn't by any mean the unique (or the most aesthetically attractive, or cleanly coded) version of the jME [31] Implementor, it is just a very simple way to demonstrate that, once an Implementor is written, many different but similar application scripts will be runnable by it and they all will presenting the features this Implementor offers, like 3D environment, inventory representation and management, interactive combinations and so on, without the need to reinvent the wheel.

The GUI interaction is handled in pure Nifty [43] style, introducing screen controllers, one for each screen. The most difficult part of the job was handling the multi-threading correctly, as Nifty [43] is not thread safe and GUI can only be updated (like scene-graph, though) at precise moments during execution; moreover, multi-threaded event actions were needed because requests to the SIMs are locking when they need to receive user input, while on the other side the graphic engine process must not be locked otherwise there is no way the user can interact with the GUI. This problem was solved defining all *NiftyUpdatableIntf* implementor classes to be registered at any time and then, at the right time, called to process the updates to the GUI. The best approach to achieve this results is a *Runnable* instanced inline, as proposed in Code Snippet 5.8:

Code Snippet 5.8: Nifty GUI Multithreading

```java
public abstract class GUIBuilder implements Runnable, NiftyUpdatableIntf {

        @Override
        public void onUpdate(float tpf) {
                run();
                implementor.unregisterNiftyUpdatable(this);
        }

}

...

GUIBuilder builder = new GUIBuilder() {
        public void run() {
                // Do stuff...
        }
};
implementor.registerNiftyUpdatable(builder);
```

Unfortunately, this pattern was not followed strictly along the code, but this would be the best solution when writing a real jME [31] Implementor for JIVES.

5.2.4 Application Layer

The Application layer of the JIVES framework has been designed with the idea to offer a new smooth and interactive way to write 3D Virtual Environment applications: this is possible through the use of the JiveScript scripting language. JiveScript has been introduced in the JIVES framework to let the developer define efficiently its own 3D Virtual Environment, specifying the whole logic of the application, the virtual objects, their relations and behaviours.

JiveScript

Being Java [51] based, there is no problem in writing a Java [51] application that relies on JIVES, compile it, and execute it. The proposed alternative, in any case, is to stick to VESLOM in coding all the necessary implementors, then write the final applications directly in JiveScript. The advantage of this approach is the internal management of scripted applications, because it's guaranteed that every participant of a session is executing the same application, and the intrinsic exploitation of the reusability of the middleware, so that a lot of different working application scripts can rely on a small set of good-written implementors. This vision derives from the fact that learning a strict syntax interpreted language like JiveScript is much more easy than entering the low level details of the system in order to hook in the application code.

JiveScript is based on Sun's *javax.script* package [52] from the JDK. This choice was preferred over Mozilla Rhino [40] not to introduce another dependency in the project and because of their built-in management of security when it comes to accessing private or protected scopes - while in Rhino [40] it's not impossible, it's harder to achieve. A trade-off in this solution is that Sun's *javax.script* package [52] does not give the full access to native JavaScript object and, where needed, this access is obtained through the Java Reflection API.

The JiveScript, indeed, can act in a sandbox mode where there is no restriction on input commands and it's possible to exploit all the power of the scripting language to create and modify JIVES objects and interact with them. This behaviour can be inhibited by the *__scripting()* directive.

A comfortable way to write JiveScript code is by mean of the NetBeans JiveScript plugin [15]. It features the same rapid writing features of the JiveScript Shell but it's fully integrated with the NetBeans [56] IDE; it provides a textual editor for **.jives* files, assuming that the project classpath is appropriately configured with the JIVES library and all the necessary Implementors.

Chapter 6

Development and Usage

6.1 Developer's Manual

JIVES is Open Source software. Anyone could branch it or contribute to the baseline, as long as all the changes made are subject to the GNU General Public License 3.0 (GPLv3) [22]. The JIVES development server is `sourceforge.net`. There is no need to be registered users to download or browse the code and the binaries, while if you wish to contribute to the baseline, you are encouraged to contact the JIVES project developers on `sourceforge.net`.

6.1.1 Development using JIVES

Every release made publicly available for download is situated at `http://sourceforge.net/projects/jives/files/`. From this location it will be possible to download the binaries of the JIVES library and of the test Implementors. The Javadoc of the project is accessible from `http://jives.sourceforge.net/javadoc/jives/`.

The guide will propose a step by step procedure to follow in order to develop with JIVES both using Eclipse [64] and jMonkeyEngine [31] SDK, which is built upon the Netbeans [56] Platform .

There are different areas of development, here briefly explained:

- **Applications**: the most common developer will need to setup the project with the JIVES library, the right Implementors, then just focus on writing the application using JiveScript.

- **Implementors and extensions**: a developer can also configure the project only using the JIVES library and write by himself the Implementors needed and maybe even customized actions that are not available by default. This code might also be exported with a built target and made available to other projects.

- **Contributing to the Core**: new features and bug fixes can be contributed to the Core by either committing directly to the JIVES CVS or submitting a patch. For this kind of development, read further in 6.1.2.

Next, this guide will show how to setup the development application using JIVES library. If you wish to use one of the test Implementors beside the library, repeat exactly the same steps shown below using the Implementor binaries instead.

Setup guide for Eclipse Indigo

1. From the File menu, select New, then Java project... and follow the on-screen instruction to create it.

2. Download the JIVES project library and extract it in the project folder. The final result should be similar to Figure 6.1.

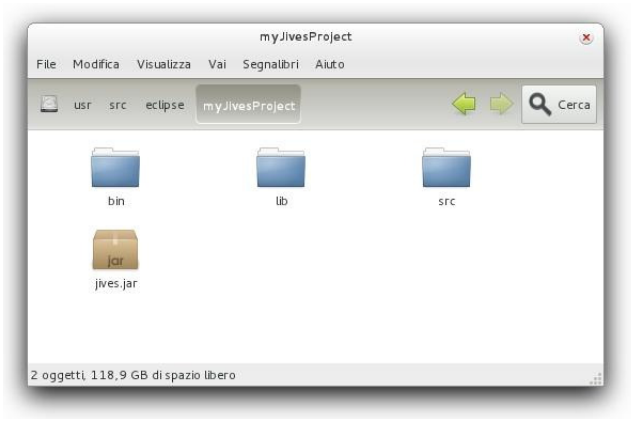

myJivesProject
File Modifica Visualizza Vai Segnalibri Aiuto
usr src eclipse myJivesProject
Cerca
bin
lib
src
jives.jar
2 oggetti, 118,9 GB di spazio libero

Figure 6.1: JIVES Project Folder using Eclipse

3. From the Package Explorer in the Java Perspective, refresh the project then select all the newly added dependencies, as shown in Figure 6.2.

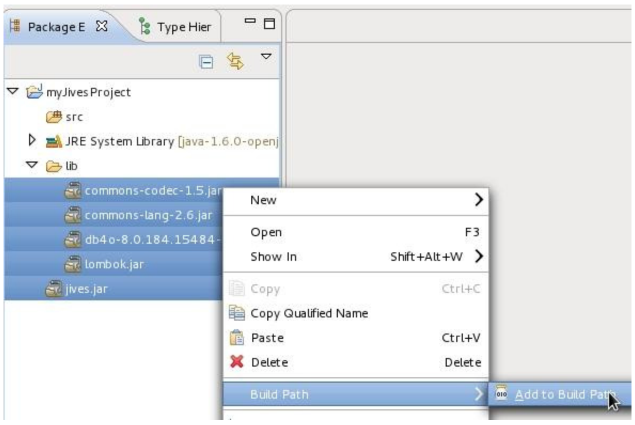


Figure 6.2: Adding JIVES Dependencies in Eclipse

4. The project is now configured to use JIVES library.

Setup guide for jMonkeyEngine SDK (Netbeans)

1. From the File menu, select New Project then follow the on-screen instruction to create a jME3 BasicGame project if you wish to write a jME [31] application, a simple Java Project otherwise.

2. Download the JIVES project library and extract it in the project folder. Figure 6.3 shows how the project folder should look like.

3. Right click on the "Library" entry on the project tree visible in the Projects tab, then select "Add JAR/Folder" and add the newly extracted dependencies to the project, as shown in Figure 6.4.

4. The project is now configured to use the JIVES Library.

Writing a JiveScript application

Even if any text processor can be used to write a JiveScript, the below procedure can be followed in order to install the JiveScript plugin [15]; the plugin will enable

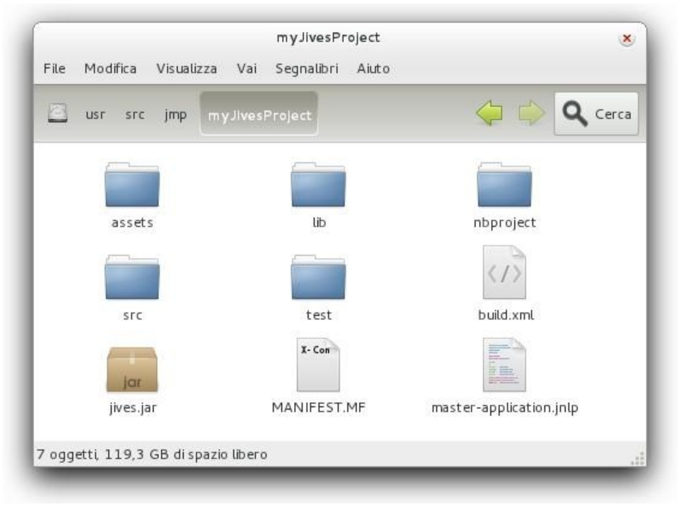

myJivesProject
File Modifica Visualizza Vai Segnalibri Aiuto
usr src jmp myJivesProject
Cerca
assets
lib
nbproject
src
test
build.xml
jives.jar
MANIFEST.MF
master-application.jnlp
7 oggetti, 119,3 GB di spazio libero

Figure 6.3: JIVES Project Folder using jME SDK

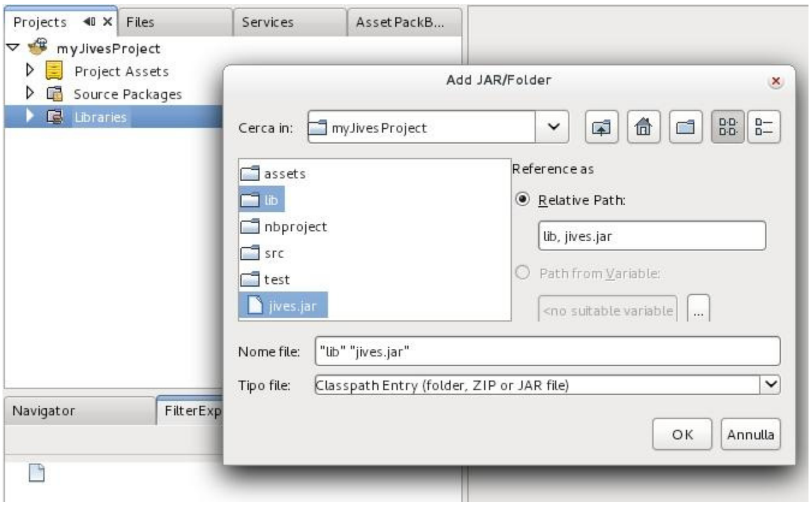


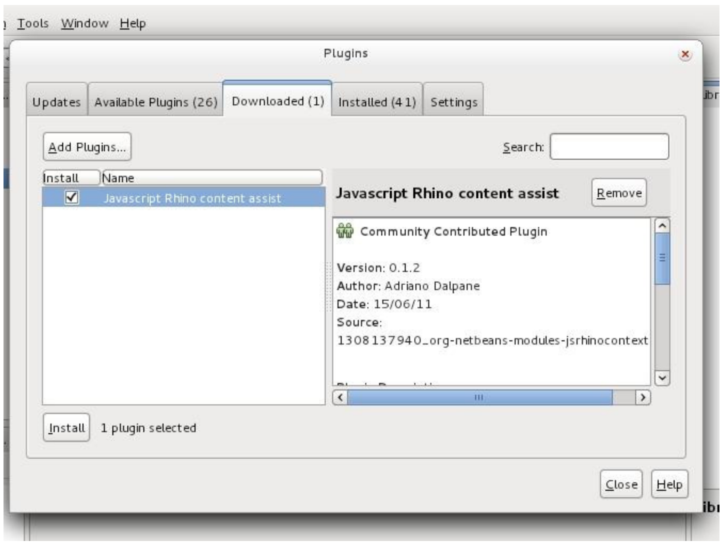
Figure 6.4: Adding JIVES Dependencies in jME SDK

content assist when editing JiveScript files and is available only for Netbeans [56] and jMonkeyEngine [31] SDK.

1. Download Rhino [40] content assist Netbeans plugin from `http://plugins.netbeans.org/plugin/39133/`.

2. From the IDE Tools menu, select Plugins. On the Downloaded tab, click the Add Plugins... button.

3. Select the downloaded file. It will be added to the list of downloaded plugins, as reported in Figure 6.5. When ready, press the Install button, follow the installation instructions and restart the IDE.



Javascript Rhino content assist

☑ Javascript Rhino content assist

Community Contributed Plugin

Version: 0.1.2
Author: Adriano Dalpane
Date: 15/06/11
Source:
1308137940_org-netbeans-modules-jsrhinocontext

1 plugin selected

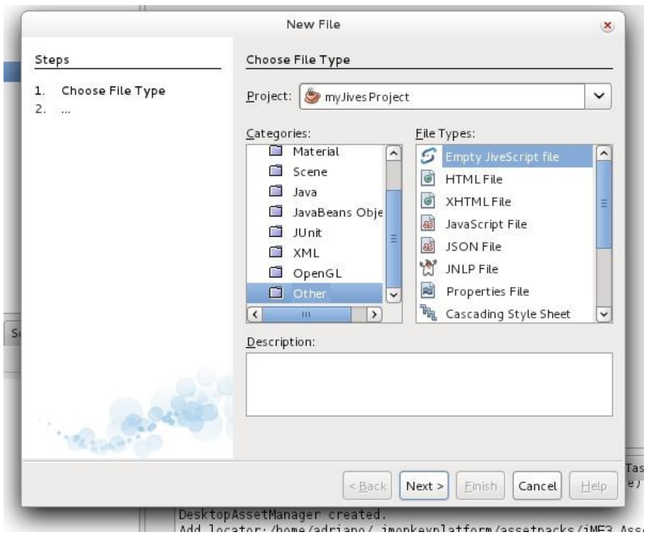
Figure 6.5: Installing JavaScript Rhino Content Assist

4. Download the JiveScript editor Netbeans plugin [15].

5. Install the JiveScript editor Netbeans plugin [15] following the same procedure shown at steps 1,2,3 for the Rhino content assist Netbeans plugin installation.

6. From the Projects tab, right click on the previously configured project name

(myJivesProject in this example) then select New, Other. The window depicted in Figure 6.6 will appear.



```
DesktopAssetManager created.
```

Figure 6.6: Creating a new JiveScript

7. Select Empty JiveScript file from the Other category. Proceed until the creation is finished. When done, the new file will be enabled to be edited using the JiveScript Editor as in Figure 6.7; restart and check that the plugin is active if this doesn't happen immediately.

6.1.2 Building the JIVES project

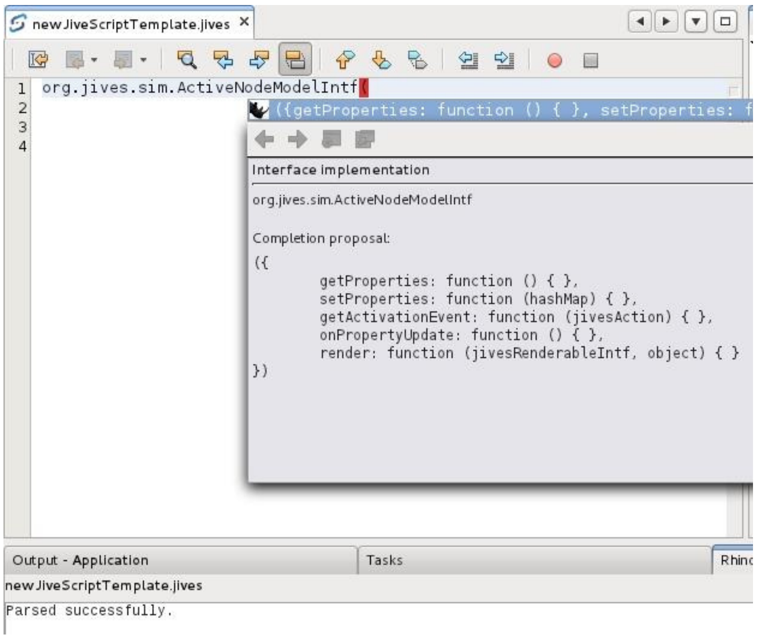
Build guide for Eclipse Indigo

1. From the File menu, select New, then Other...

2. In the CVS category, select Projects from CVS, then click next. Configure the connection as reported in Figure 6.8 and proceed.

3. In the Select Module, chose the Use specified module name option and fill it with "jives".

4. Press finish to import the CVS project into the workspace.



```
org.jives.sim.ActiveNodeModelIntf(
({getProperties: function () { }, setProperties: f
```

Interface implementation

org.jives.sim.ActiveNodeModelIntf

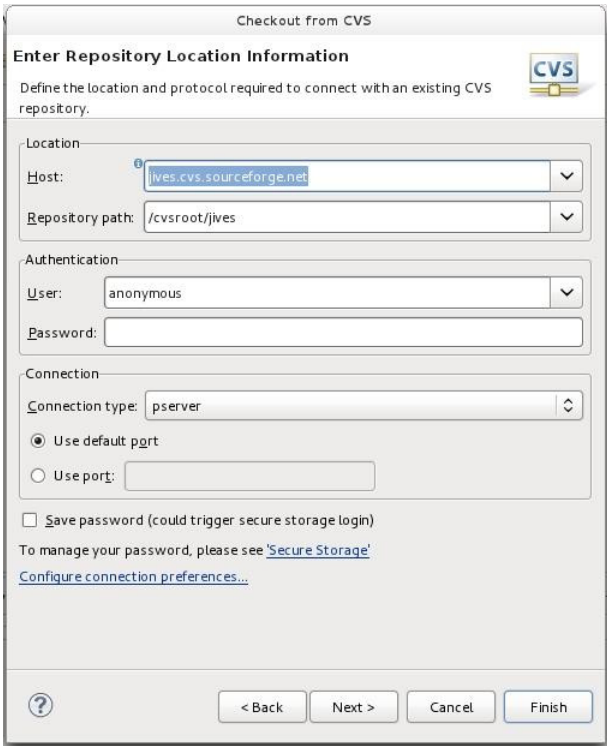
Completion proposal:

```
({
        getProperties: function () { },
        setProperties: function (hashMap) { },
        getActivationEvent: function (jivesAction) { },
        onPropertyUpdate: function () { },
        render: function (jivesRenderableIntf, object) { }
})
```

newJiveScriptTemplate.jives

```
Parsed successfully.
```

Figure 6.7: JiveScript Editor Netbeans plugin



Enter Repository Location Information

Define the location and protocol required to connect with an existing CVS repository.

Location

Host: jives.cvs.sourceforge.net

Repository path: /cvsroot/jives

Authentication

User: anonymous

Password:

Connection

Connection type: pserver

(●) Use default port

( ) Use port:

☐ Save password (could trigger secure storage login)

To manage your password, please see 'Secure Storage'

Configure connection preferences...

Figure 6.8: Configuring the CVS connection in Eclipse

Build guide for jMonkeyEngine SDK (Netbeans)

1. From the Team Menu, select CVS then Checkout...
2. Specify the following connection string:
   :pserver:anonymous@jives.cvs.sourceforge.net/cvsroot/jives.
3. In the Module to Checkout section, write "jives", then click Finish to import the project into the workspace.

Build targets

Once the project is setup in your favourite IDE, you can use the predefined build targets to generate the different parts of the JIVES project. Those are:

- **clean**: cleans the output directory.
- **jives (default)**: builds the JIVES library.
- **implementor-engine-shell**: builds the *ShellImplementor*.
- **implementor-engine-jme**: builds the *JMEImplementor*.
- **implementor-network-jxse**: builds the *JXSEImplementor*.
- **testNetworkApp**: builds a test application to run a JXSE Peer in SWING.
- **testShell**: builds a test application that uses the *ShellImplementor* and the *JXSEImplementor*.
- **testJME**: builds a test application that uses the *JMEImplementor* and the *JXSEImplementor*.

Moreover there is the possibility to export the *JMEImplementor* directly as an applet using jMonkeyEngine [31] SDK, so that it can run in the web browser. You just need to be sure that the project is configured to be built as an applet.

This can be done in the project properties window, Application section, Applet subsection: Create Applet has to be checked. The appropriate class to run as an applet can be found in the *org.jives.test* package and it's called *TestJMEApplet*. The peculiarity of this class is that it extends *JMEImplementor*, in order to run as an applet, as can be seen in Code Snippet 6.1.

Code Snippet 6.1: TestJMEApplet

```java
public class TestJMEApplet extends JMEImplementor {

    public TestJMEApplet() {
        super();
    }

    @Override
    public void simpleInitApp() {
        Log.setLogLevel(Log.LOG_ERROR);

        instance = this;
        Jives.setEngine(instance);
        Jives.setNetwork(JXSEImplementor.getInstance());

        super.simpleInitApp();

        // Run Jives loop in another thread
        new Thread() {
            @Override
            public void run() {
                Jives.getEngine().loop();
                Jives.getNetwork().stopNetwork();
                System.exit(0);
            }
        }.start();
    }
}
```

At the moment of writing the procedure that automatically builds the applet is not completely working, thus it must be integrated with the following BASH script (Code Snippet 6.2) (works on Mac OS X, too).

Code Snippet 6.2: Applet builder Bash Script

```
#!/bin/bash
dir='/usr/src/jmp/JivesApplet'
cd $dir/dist/Applet
mkdir code
mv code.jar code
cd code
jar -xf code.jar
rm './META-INF/BCKEY.SF' './META-INF/BCKEY.DSA' 'code.jar'
cp -R $dir/assets/* .
jar cf ../code.jar *
cd ~
jarsigner -keystore .jivesks $dir/dist/Applet/code.jar adriano
rm -R $dir/dist/Applet/code
```

The steps performed by the script are the following:

1. Sets the project directory. It must point to the location where the applet under development is present.

2. Removes duplicates from the *jar* file beside removing an invalid signature and copying the ”assets” directory.

3. Repackages the *jar*.

4. Signs the *jar*, assuming that exists a *~/.jivesks* file that represent a keystore to use for the sign and the user (”adriano” in the above script) is able to sign it.

5. Removes the temporary extraction directory.

After a clean build and the execution of the script the JME test is working as an applet, as shown in Figure 6.9.

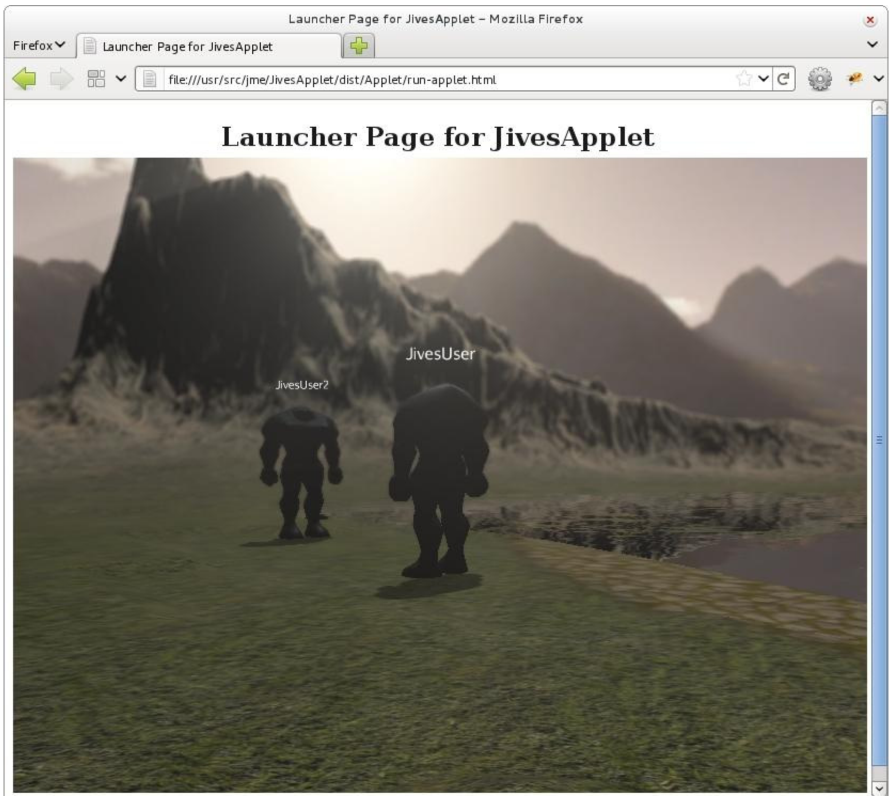

Launcher Page for JivesApplet – Mozilla Firefox
Firefox
Launcher Page for JivesApplet
file:///usr/src/jme/JivesApplet/dist/Applet/run-applet.html
Launcher Page for JivesApplet
JivesUser
JivesUser2

Figure 6.9: Executing the JivesApplet

6.2 User's Manual

The JIVES framework, at the moment, lets the developer create both textual and 3D Virtual Environment applications. Two different User's Manuals are proposed: the first one related to applications built using the Shell Implementor, the second one related to applications built using the jME [31] Implementor. Obviously, according to the different Implementors that a developer can create for the JIVES framework, totally different User's Manuals must be created. The Shell User's Manual and the jME [31] User's Manual proposed in this section teach the user how to start and use applications built with the Implementors provided by default in the JIVES framework. A third User's Manual is related to how start a JIVES application in a Web Environment.

In order to run a JIVES Virtual Environment application, the user needs a Java Virtual Machine installed. If the user does not have a Java Virtual Machine installed, he just needs to visit the Java SE Downloads official page [50] and download the Java Runtime Environment (JRE) or the Java Development Kit (JDK) setup packages for his operating system and then install the downloaded file. The sample JIVES applications have been successfully tested by using the Java SE 6 Update 29 Development Kit. See the Java [51] SE Documentation [49] for further informations about installing the JVM.

When JIVES has been executed, it creates automatically a hidden folder called ".jives" in the home directory of the user's OS. In a MAC OS X or Linux Environment the folder is located in *$userHome*; while in a Windows system the folder is named "Roaming\jives" and is created in *C:\Documents and Settings\$user\Application Data\Roaming\jives* or *C:\Users\$user\AppData\Roaming\jives* depending on the Windows version; the folder ".jives" ("Roaming\jives") contains two sub-folders: one is called "config", the another one "states". The "config" directory contains a subfolder called "jxse" in which are placed the default JIVES network configuration file *networkConfiguration.default.xml* and the customizable *networkConfiguration.xml* network configuration file. The latter file includes all the custom network settings of the user who is running the JIVES application. The "states" directory will contain all the user save files. The "states" folder is created only when there is at least one save file. The folder disposal is shown in Figure 6.10.

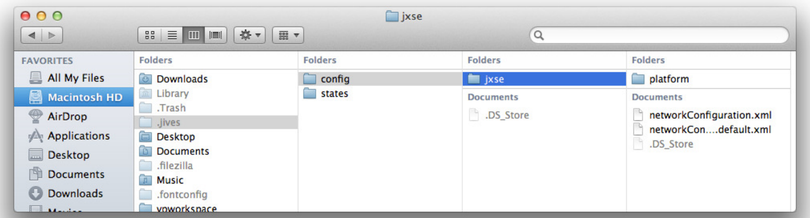
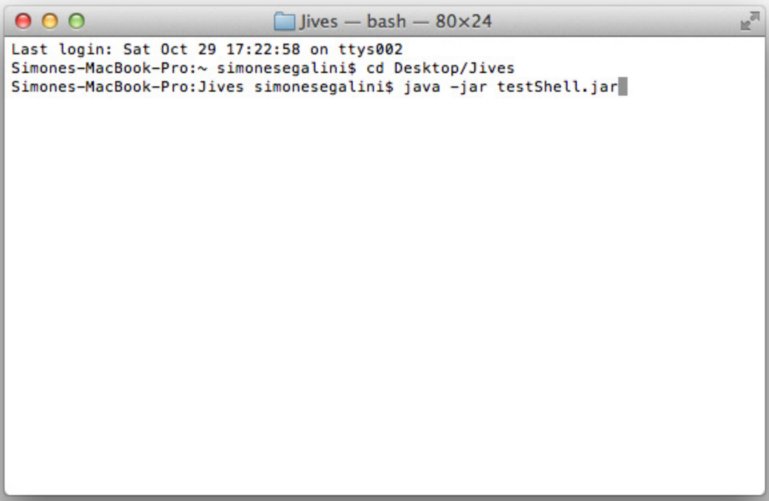


Figure 6.10: JIVES Folder Disposal

Although JIVES creates the ".jives" ("Roaming\jives") folder and its subfolders automatically, it is still acceptable that the user creates them manually and inserts in the right location the network settings configuration files. Differently from Windows, in which the *Roaming\jives* folder is visible; the ".jives" directory created in MAC OS X and Linux systems is a hidden folder. In order to access the hidden files and directories in a MAC OS X Environment, the user has to open a terminal and launch the command *defaults write com.apple.finder AppleShowAllFiles -bool true*, then restarting the Finder by executing the command *killall finder*; while in a Linux system it depends on the specific distribution.

6.2.1 Shell User's Manual

A pre-compiled JIVES application is composed by a single *jar* file, the archive file format typically used to aggregate many Java class files and associated metadata and resources into one file to distribute application software or libraries on the Java [51] platform. Each JIVES application built by using the Shell Implementor runs over a terminal, a shell or a console. Under a MAC OS X operating system, the terminal is accessible in the directory */Applications/Utilities/Terminal*. Once the Terminal has been started, the user has to start the JIVES application by reaching the right directory where the application is located. In order to explore the directories in the Terminal, the user has to use the command *cd* followed by the name of the folder in which he desires to move. For example, if the pre-compiled *jar* file is located in a folder called "Jives" on the Desktop, the user has to type in the Terminal the command *cd Desktop/Jives/*. Once the Terminal points to the desired folder, the user has only to start JIVES by launching the command *java -jar testShell.jar*, where "testShell" is the name of the *jar* file, as shown in Figure 6.11. The procedure is successfully tested using a MAC OS X Lion operating system.

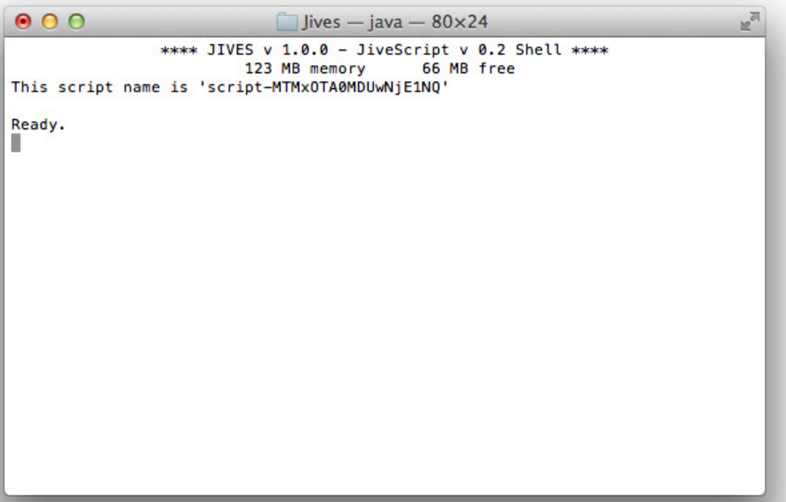


```
Last login: Sat Oct 29 17:22:58 on ttys002
Simones-MacBook-Pro:~ simonesegalini$ cd Desktop/Jives
Simones-MacBook-Pro:Jives simonesegalini$ java -jar testShell.jar
```

Figure 6.11: Starting JIVES from Mac OS X Terminal

Once JIVES has been started, the Terminal appears as shown in Figure 6.12.

Under a Linux operating system, the procedure is quite similar to the one proposed for MAC OS X. Because the number of Linux distributions is quite huge, there is no reason to explain here how to start a shell or terminal for each



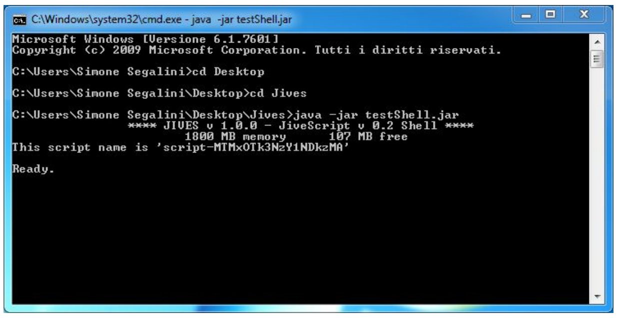
```
                    **** JIVES v 1.0.0 - JiveScript v 0.2 Shell ****
                           123 MB memory      66 MB free
This script name is 'script-MTMxOTA0MDUwNjE1NQ'

Ready.
```

Figure 6.12: Shell Implementor Starting Screen

Linux distribution. Once a terminal has been started, the commands the user has to type are the same ones used in MAC OS X: the *cd* command must be used to reach the JIVES directory and the command *java -jar jives.jar* to launch JIVES. The procedure has been successfully tested using a Linux Fedora 15 distribution, virtualized by Parallels Desktop in a MAC OS X Lion Environment.

Under a Windows operating system, the procedure is almost the same. The user has just to click the *Start* button, then digit the *cmd* command under the *run* option. This command executes the Command Prompt, a terminal for Windows operating systems. Once the Prompt has been started the user has to move to the right directory in which JIVES is located using the *cd* command and then execute JIVES typing the command *java -jar testShell.jar*, where "testShell" is the name of JIVES application file. The whole procedure is reported in Figure 6.13. This procedure has been successfully tested using a Windows Seven Professional 64 bit operating system.

When running a JIVES Shell Virtual Environment application, in order to



```
Microsoft Windows [Versione 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Tutti i diritti riservati.

C:\Users\Simone Segalini>cd Desktop

C:\Users\Simone Segalini\Desktop>cd Jives

C:\Users\Simone Segalini\Desktop\Jives>java -jar testShell.jar
                **** JIVES v 1.0.0 - JiveScript v 0.2 Shell ****
                        1800 MB memory      107 MB free
This script name is 'script-MTMxOTk3NzY1NDkzMA'

Ready.
```

Figure 6.13: Starting JIVES from Windows Command Prompt

set the network configuration, the user has to set it manually. This means that the user has to open with any text editor the *networkConfiguration.xml* file located in the sub-folder "jxse" of the folder "config". The first time that the user runs the application, it happens that there is not the ".jives" ("Roaming\jives" in Windows) folder and consequently there are not also the *networkConfiguration.default.xml* and the *networkConfiguration.xml* files. If the user has no interest to change the network configuration, he has only to start the JIVES application: JIVES itself will create the *networkConfiguration.xml* file for the user. If the user wants to modify some properties with respect to the default configuration, he has to start JIVES at least once, even without loading any script: in that way the Networking Layer Implementor will create the ".jives" ("Roaming\jives") folder and the necessary network configuration files in the predefined location. The user has only to modify the content of *networkConfiguration.xml* with a text editor. The *networkConfiguration.xml* file can be even modified at runtime (but before loading a script): once set the network parameters, it is sufficient to stop the network (if a script has been previously loaded) by launching the *reset()* command and load a new JiveScript with the *load()* command: when a new JiveScript is loaded the new settings are read and stored and the network restarted. In Code Snippet 6.3 is shown the default network configuration provided by JIVES.

Code Snippet 6.3: JIVES Network Configuration Default Settings

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<network>
	<jives>
		<jivesUsername>JivesUser</jivesUsername>
	</jives>
	<user>
		<proxyHost/>
		<proxyPort/>
		<proxyUsername/>
		<proxyPassword/>
	</user>
	<server>
		<urlHost>http://jives.sourceforge.net/rendezvous/</urlHost>
		<urlSecretKey>http://jives.sourceforge.net/access/getSecretKey.php</urlSecretKey>
		<httpsUsername></httpsUsername>
		<httpsPassword></httpsPassword>
	</server>
	<lan>
		<rendezvous_ipv4/>
		<rendezvous_ipv4_port/>
		<rendezvous_ipv6/>
		<rendezvous_ipv6_port/>
	</lan>
	<interface>
		<netInterface>lo</netInterface>
	</interface>
	<internet_lan>
		<choice>lan</choice>
		<proxy>false</proxy>
		<ipv6>false</ipv6>
	</internet_lan>
</network>
```

The user has the possibility to modify the name of his avatar in the Virtual Environment by modifying the *jivesUsername* field, or setup the kind of connection by typing in the *choice* field ”lan” if the user wants to execute the application in a LAN network, or ”internet” if the user wants to use the Internet connection. In the first case, the LAN connection, the user has the possibility to choose if he wants to start the JIVES application assuming a Rendezvous role or not. If the user wants to start as a Rendezvous he does not have to add anything else; at the contrary if he wants to start as a simple Peer and connect to a Rendezvous obviously located in the same LAN network, he has to specify the *rendezvous_ipv4* and *rendezvous_ipv4_port* fields filling them with the IPv4 and the port of the Rendezvous to which he wants to connect. For example, is the Rendezvous has the IPv4 *192.168.0.8* and the Port *9701* the right way to configure the network settings configuration file is to change the *rendezvous_ipv4* field in *<rendezvous_ipv4>192.168.0.8</rendezvous_ipv4>* and the *rendezvous_ipv4_port* field in *<rendezvous_ipv4_port>9701</rendezvous_ipv4_port>*. In the case he wants to use an IPv6 connection, the user has to enable it by setting to "true" the *ipv6* field, and if he roles as a simple Peer in a LAN network the fields to be correctly set in this case are the *rendezvous_ipv6* and *rendezvous_ipv6_port* ones. The IPv6 connection feature provided by JIVES is still an experimental feature, and may have some issues in its proper functioning, due to the lack of IPv6 connections managed by Italian providers and IPv6-enabled routers.

A very important field the user has to set is the *netInterface* field. If this field remains unchanged, the user can execute the JIVES application only using the *loopback interface*, so the connection works only locally (on the same hardware machine without having the possibility to run JIVES in a LAN or in the Internet): any traffic that a JIVES application sends to the *loopback interface* is immediately received on the same interface. The user has to change the *netInterface* field by specifying the name of the interface through which it is possible to use a working connection. For example, if the enabled working inteface is called "en1" the *netinterface* field has to be set in that way: *<netInterface>en1</netInterface>*. In order to discover which is the working interface running on the user's computer, it is sufficient to run the command *ifconfig* in a terminal under a Linux or MAC OS X Environment, or the command *ipconfig* if the OS is Windows-based. These commands will show to the user all the network interfaces available on his computer: the right one is the interface which is correctly connected to a LAN network or Internet and to which the router has assigned a valid IPv4 or IPv6.

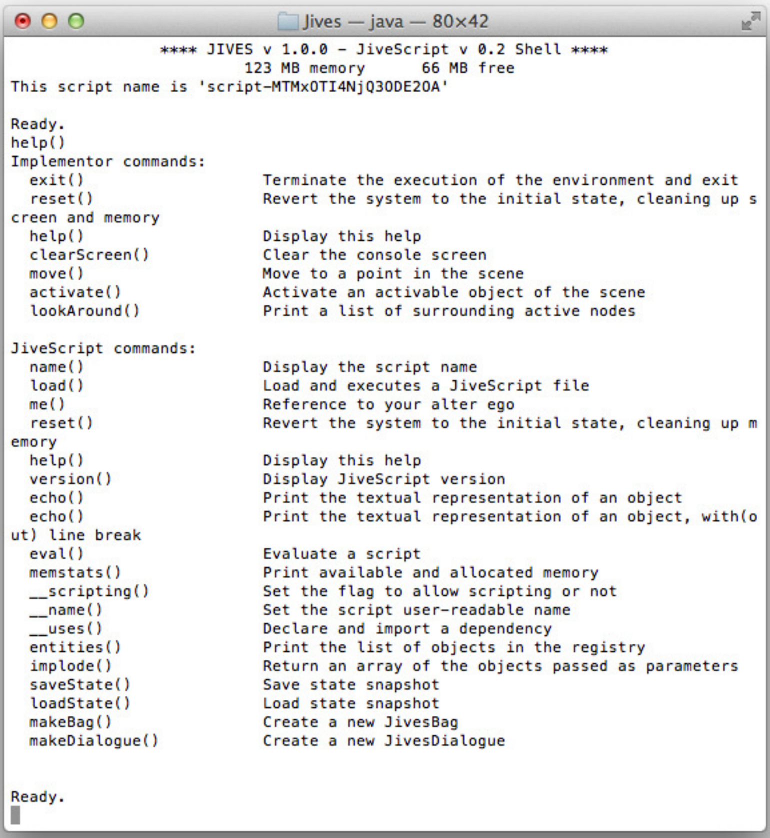
All the fields belonging to the *server* tag are related to the server in which is hosted the directory of Rendezvous. These fields are useful only if the user wants to execute the JIVES application using an Internet connection. In order to show the proper functioning of the Demo using an Internet connection, a Rendezvous directory is hosted on the Sourceforge [24] JIVES web space. Obviously the developer has to provide the correct url to the user or set it as the default configuration setting. The *urlSecretKey* field is related to the account identification: each account is associated to a secret key known only to the system; it is used on the account behalf to perform important operations like storing the application state. This field has been set by default to a web page hosted by Sourceforge [24] in order to show the proper functioning of this feature when running the Demo. Also in this case the JIVES developer has to provide to the user the correct url or set it by default in the *networkConfiguration.default.xml* file. The last two fields of the *server* tag are related to the account login on the server using an HTTPS connection to the *urlSecretKey* mentioned before.

Another feature that can be set modifying the *networkConfiguration.xml* file is the possibility to connect to the Internet also when the connection works under a Proxy. In this case the user has to set to "true" the *proxy* field and define all the fields under the *user* tag: the url of the Proxy server, the port to which the user has to connect, the username and the password to establish a working connection to the Proxy server.

Sample Shell JiveScript

In order to show the proper functioning of a JIVES Virtual Environment application built using the Shell Implementor, a Shell Demo has been developed. The whole logic of the Demo has been written directly by using the JiveScript language: the user has just to start JIVES and load the right JiveScript. In order to correctly load the script, the file must have the "jives" extension. Once the user has started JIVES in a console environment, some commands are available to perform specific operations. These commands can be shown by typing in the console the command *help()*.

As shown in Figure 6.14 the command *help()* let the user see two kinds of commands: the first ones are specifically related to the Shell Implementor and



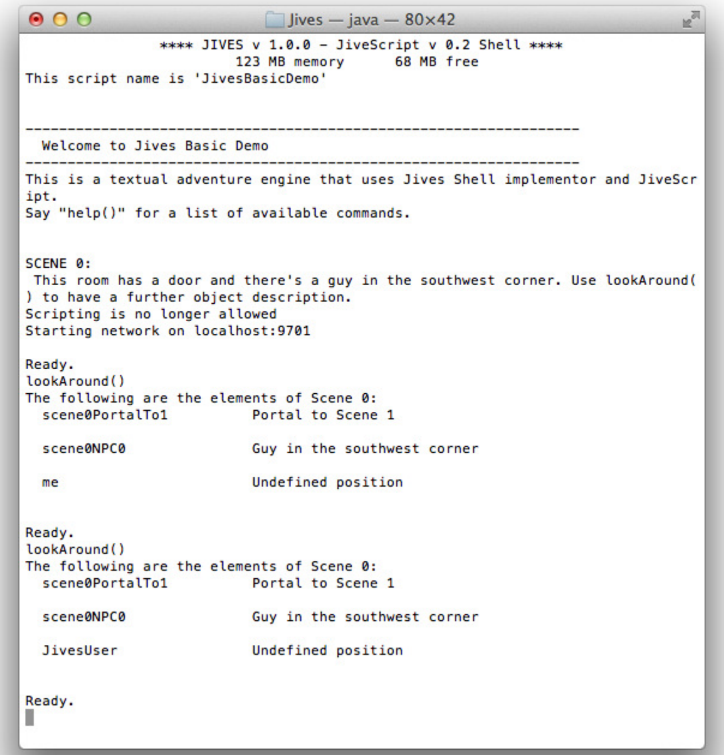
```
                **** JIVES v 1.0.0 - JiveScript v 0.2 Shell ****
                         123 MB memory      66 MB free
This script name is 'script-MTMxOTI4NjQ3ODE2OA'

Ready.
help()
Implementor commands:
  exit()                    Terminate the execution of the environment and exit
  reset()                   Revert the system to the initial state, cleaning up s
creen and memory
  help()                    Display this help
  clearScreen()             Clear the console screen
  move()                    Move to a point in the scene
  activate()                Activate an activable object of the scene
  lookAround()              Print a list of surrounding active nodes

JiveScript commands:
  name()                    Display the script name
  load()                    Load and executes a JiveScript file
  me()                      Reference to your alter ego
  reset()                   Revert the system to the initial state, cleaning up m
emory
  help()                    Display this help
  version()                 Display JiveScript version
  echo()                    Print the textual representation of an object
  echo()                    Print the textual representation of an object, with(o
ut) line break
  eval()                    Evaluate a script
  memstats()                Print available and allocated memory
  __scripting()             Set the flag to allow scripting or not
  __name()                  Set the script user-readable name
  __uses()                  Declare and import a dependency
  entities()                Print the list of objects in the registry
  implode()                 Return an array of the objects passed as parameters
  saveState()               Save state snapshot
  loadState()               Load state snapshot
  makeBag()                 Create a new JivesBag
  makeDialogue()            Create a new JivesDialogue


Ready.
█
```

Figure 6.14: List of Shell Implementor and JiveScript Commands

they may change according to the Implementor the JIVES application is running; the second ones are related to the JiveScript language. The first operation the user has to perform once JIVES has been started is obviously load the JiveScript Shell Demo. This is possible by typing the command *load("file:/path of the JiveScript Shell Demo").* Note that the user has to specify the absolute path of the JiveScript Demo. The first part of the argument of the *load* command (*file:/*) represents the protocol used by JIVES to load the file; the second part is the path of the location in which is located the script. For example, in a MAC OS X environment, if the JiveScript Demo has been located on the Desktop, the user has to type *load("file:/Users/$user/Desktop/basic-demo-shell.jives")* where $user is the name of the operating system user. When loading a script, Windows users has to specify the path using the double backslash expression: with respect to the previous example, in a Windows environment the *load* command becomes *load("file:/C:\\Users\\$user\\Desktop\\basic-demo-shell.jives").* Once the script has been loaded the terminal shows a screen as in Figure 6.15.

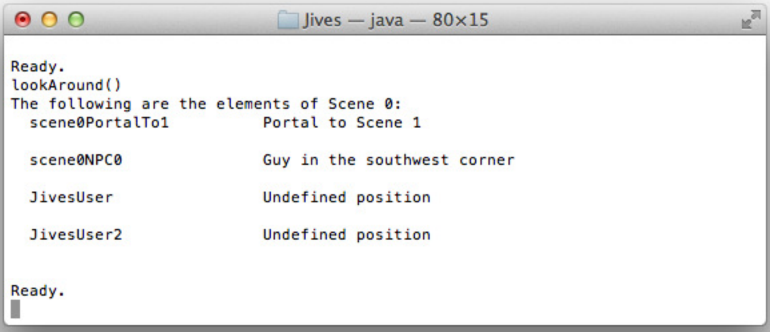
The user is located in the starting scene. Using the *lookAround()* command it is possible to see the elements presents in the scene: the Non-Playing Characters (NPCs), the Playing Characters (PCs) and the Artifacts, such as Portals that permit to move from a scene to the next one or vice versa. In Figure 6.15 there is only the player called "JivesUser" in the scene, but if an other user runs the same JIVES application it is possible to see the new entry by retyping the *lookAround()* command, as shown in Figure 6.16.

From the point of view of the second user, in order to connect to the same JIVES Virtual Environment application that the first user is running, it is necessary to perform the usual operations needed to launch a JIVES application. In particular, the script has to be exactly the same that the first user is running, while the network configuration is different. If the JIVES Virtual Environment application works on a LAN connection, the second user has to specify in the



```
                 **** JIVES v 1.0.0 - JiveScript v 0.2 Shell ****
                           123 MB memory      68 MB free
This script name is 'JivesBasicDemo'


-------------------------------------------------------------------
  Welcome to Jives Basic Demo
-------------------------------------------------------------------
This is a textual adventure engine that uses Jives Shell implementor and JiveScr
ipt.
Say "help()" for a list of available commands.


SCENE 0:
 This room has a door and there's a guy in the southwest corner. Use lookAround(
) to have a further object description.
Scripting is no longer allowed
Starting network on localhost:9701

Ready.
lookAround()
The following are the elements of Scene 0:
  scene0PortalTo1           Portal to Scene 1

  scene0NPC0                Guy in the southwest corner

  me                        Undefined position


Ready.
lookAround()
The following are the elements of Scene 0:
  scene0PortalTo1           Portal to Scene 1

  scene0NPC0                Guy in the southwest corner

  JivesUser                 Undefined position


Ready.
```

Figure 6.15: JiveScript Shell Demo Starting Scene



```
Ready.
lookAround()
The following are the elements of Scene 0:
  scene0PortalTo1           Portal to Scene 1

  scene0NPC0                Guy in the southwest corner

  JivesUser                 Undefined position

  JivesUser2                Undefined position


Ready.
```

Figure 6.16: A New Entry in the Scene

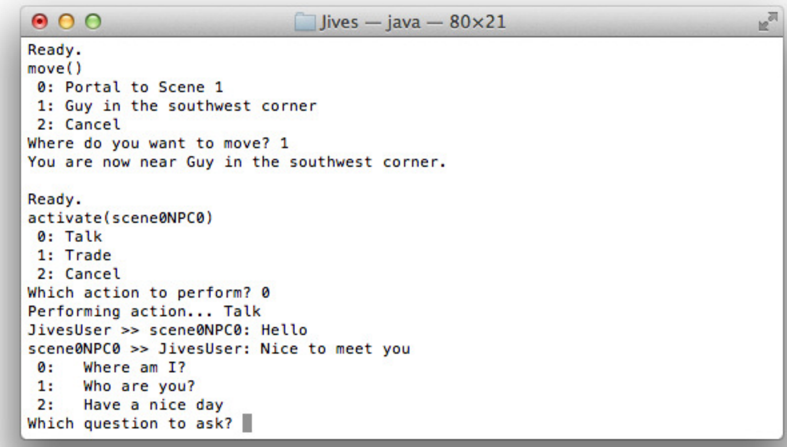
networkConfiguration.xml file the IPv4 and the port used by the first user that acts as a Rendezvous, as shown in Code Snippet 6.4.

Code Snippet 6.4: JIVES Peer Network Configuration Settings

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<network>
        <jives>
                <jivesUsername>JivesUser2</jivesUsername>
        </jives>
        <user>
                <proxyHost/>
                <proxyPort/>
                <proxyUsername/>
                <proxyPassword/>
        </user>
        <server>
                <urlHost>http://jives.sourceforge.net/rendezvous/</urlHost>
                <urlSecretKey>http://jives.sourceforge.net/access/getSecretKey.php</urlSecretKey>
                <httpsUsername></httpsUsername>
                <httpsPassword></httpsPassword>
        </server>
        <lan>
                <rendezvous_ipv4>192.168.0.5</rendezvous_ipv4>
                <rendezvous_ipv4_port>9701</rendezvous_ipv4_port>
                <rendezvous_ipv6/>
                <rendezvous_ipv6_port/>
        </lan>
        <interface>
                <netInterface>en1</netInterface>
        </interface>
        <internet_lan>
                <choice>lan</choice>
                <proxy>false</proxy>
                <ipv6>false</ipv6>
        </internet_lan>
</network>
```

By default, the port used by JXSE [54] is the *9701* port, but it can happen that a Rendezvous start on one of the subsequent ports, such as *9702* or *9703*, and so on. In this case the user who acts as Rendezvous has the responsibility to communicate the right port to the user who acts as a simple Peer. If the JIVES Virtual Environment application is running by using a Internet connection, the second user does not have to specify the IPv4 and the port of the Rendezvous because he is not able to know them in advance: JIVES automatically queries the Rendezvous directory on the server and provides the right informations in order to connect to a Rendezvous running the same JIVES Virtual Environment application around the world.

As shown in Figure 6.16, now in the scene there are two PCs, one NPC and the Portal that lets switch from the current scene to the next one. The user can move around the scene by using the *move()* command, as shown in Figure 6.17, and choose the destination of his movement.

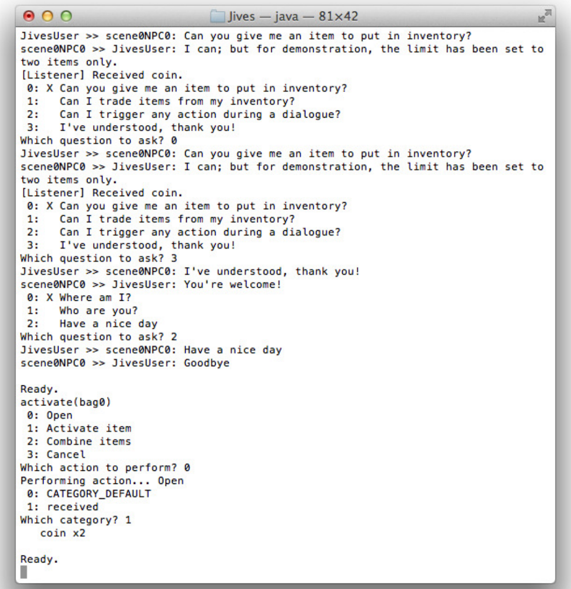


```
Ready.
move()
 0: Portal to Scene 1
 1: Guy in the southwest corner
 2: Cancel
Where do you want to move? 1
You are now near Guy in the southwest corner.

Ready.
activate(scene0NPC0)
 0: Talk
 1: Trade
 2: Cancel
Which action to perform? 0
Performing action... Talk
JivesUser >> scene0NPC0: Hello
scene0NPC0 >> JivesUser: Nice to meet you
 0:   Where am I?
 1:   Who are you?
 2:   Have a nice day
Which question to ask?
```

Figure 6.17: Moving around the Scene and Interacting with a NPC

The user can interact with the elements of the scene by using the *activate()* command, passing as argument the ID string or the local variable representing the entity he wants interact with; names that the user can retrieve by launching the *lookAround()* command. In Figure 6.17 the user interacts with a NPC element and executes one of the actions provided by the developer of the application. The JIVES Shell Demo is an Inventory-based Virtual Environment and let the user collect and combine items. The user has the possibility to perform all the operations related to the Inventory by using the *activate(bag0)* command, as shown in Figure 6.18.

Once the user has in his Inventory Bag some items, he can combine them by using the "combine" action: for each selected item the user must choose the right



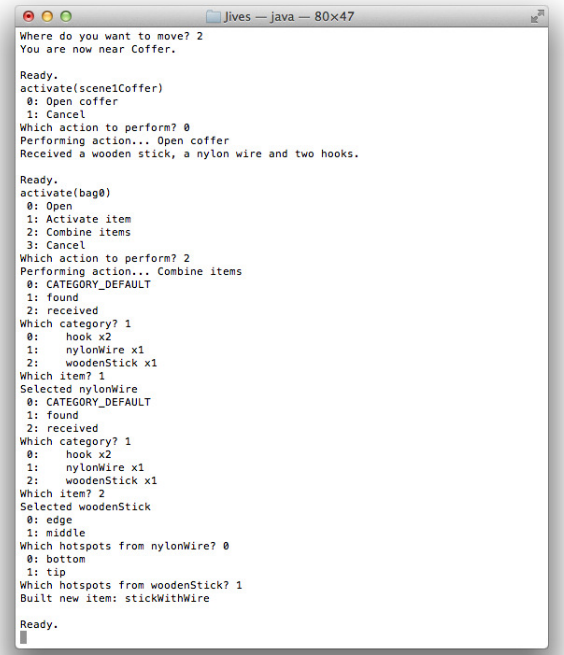
```
JivesUser >> scene0NPC0: Can you give me an item to put in inventory?
scene0NPC0 >> JivesUser: I can; but for demonstration, the limit has been set to
two items only.
[Listener] Received coin.
 0: X Can you give me an item to put in inventory?
 1:   Can I trade items from my inventory?
 2:   Can I trigger any action during a dialogue?
 3:   I've understood, thank you!
Which question to ask? 0
JivesUser >> scene0NPC0: Can you give me an item to put in inventory?
scene0NPC0 >> JivesUser: I can; but for demonstration, the limit has been set to
two items only.
[Listener] Received coin.
 0: X Can you give me an item to put in inventory?
 1:   Can I trade items from my inventory?
 2:   Can I trigger any action during a dialogue?
 3:   I've understood, thank you!
Which question to ask? 3
JivesUser >> scene0NPC0: I've understood, thank you!
scene0NPC0 >> JivesUser: You're welcome!
 0: X Where am I?
 1:   Who are you?
 2:   Have a nice day
Which question to ask? 2
JivesUser >> scene0NPC0: Have a nice day
scene0NPC0 >> JivesUser: Goodbye

Ready.
activate(bag0)
 0: Open
 1: Activate item
 2: Combine items
 3: Cancel
Which action to perform? 0
Performing action... Open
 0: CATEGORY_DEFAULT
 1: received
Which category? 1
   coin x2

Ready.
```

Figure 6.18: Managing the Item Inventory

HotSpot through which it is possible to bind the item to another one. If the combination is the one provided by the JIVES developer, the user will receive a new item obtained from the combination of the two original items. The whole procedure is shown in Figure 6.19.

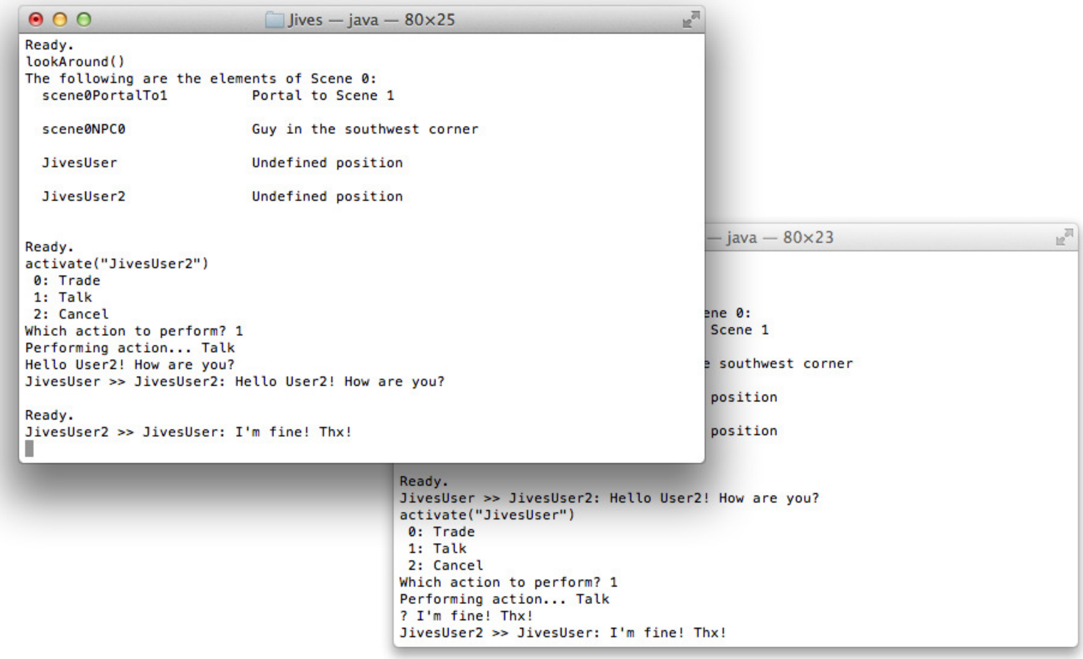
It is possible also to freely chat with one or more other users by activating them and selecting the "Talk" action. Note that in order to activate a PC, differently from the elements that are predefined in the scene, the user has to use the *activate()* command by defining the name of the PC enclosed in double quotes, because there is no local variable that refers to them. Figure 6.20 represents a chat between two PCs.

The JIVES Shell Demo let the user also trade items with NPCs and PCs: this is possible by activating the other user or the NPC with the *activate()* command and selecting the "trade" action. In the case of trading with a NPC the user has to select before requesting the trade the items that he wants to exchange and then



```
Where do you want to move? 2
You are now near Coffer.

Ready.
activate(scene1Coffer)
 0: Open coffer
 1: Cancel
Which action to perform? 0
Performing action... Open coffer
Received a wooden stick, a nylon wire and two hooks.

Ready.
activate(bag0)
 0: Open
 1: Activate item
 2: Combine items
 3: Cancel
Which action to perform? 2
Performing action... Combine items
 0: CATEGORY_DEFAULT
 1: found
 2: received
Which category? 1
 0:    hook x2
 1:    nylonWire x1
 2:    woodenStick x1
Which item? 1
Selected nylonWire
 0: CATEGORY_DEFAULT
 1: found
 2: received
Which category? 1
 0:    hook x2
 1:    nylonWire x1
 2:    woodenStick x1
Which item? 2
Selected woodenStick
 0: edge
 1: middle
Which hotspots from nylonWire? 0
 0: bottom
 1: tip
Which hotspots from woodenStick? 1
Built new item: stickWithWire

Ready.
```

Figure 6.19: Combining Two Items through HotSpots

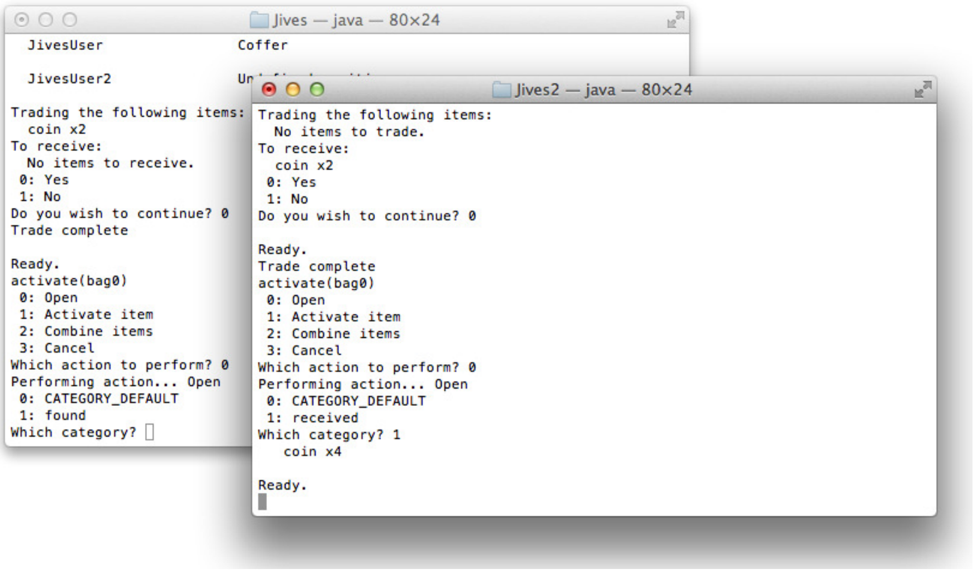


```
Ready.
lookAround()
The following are the elements of Scene 0:
  scene0PortalTo1          Portal to Scene 1

  scene0NPC0               Guy in the southwest corner

  JivesUser                Undefined position

  JivesUser2               Undefined position


Ready.
activate("JivesUser2")
 0: Trade
 1: Talk
 2: Cancel
Which action to perform? 1
Performing action... Talk
Hello User2! How are you?
JivesUser >> JivesUser2: Hello User2! How are you?

Ready.
JivesUser2 >> JivesUser: I'm fine! Thx!
```

```
ene 0:
 Scene 1

e southwest corner

 position

 position

Ready.
JivesUser >> JivesUser2: Hello User2! How are you?
activate("JivesUser")
 0: Trade
 1: Talk
 2: Cancel
Which action to perform? 1
Performing action... Talk
? I'm fine! Thx!
JivesUser2 >> JivesUser: I'm fine! Thx!
```

Figure 6.20: Free Chat between two Playing Characters

proceed with the exchange. In case of trading with a PC, the user specifies which items wants to trade and in which quantity, and sends a request to the other user; the latter has the possibility to accept or reject the trade and select one or more items to trade, or even no one. Once both the traders agreed on the exchange, the trade is performed. A trade between two PCs is shown in Figure 6.21: at the beginning both the traders have two *coins*; at the end of the trade the second user has received the two *coins* of the first one coming to have four *coins*, while the latter no longer has any item.



```
  JivesUser                 Coffer

  JivesUser2                Un

Trading the following items:
  coin x2
To receive:
  No items to receive.
 0: Yes
 1: No
Do you wish to continue? 0
Trade complete

Ready.
activate(bag0)
 0: Open
 1: Activate item
 2: Combine items
 3: Cancel
Which action to perform? 0
Performing action... Open
 0: CATEGORY_DEFAULT
 1: found
Which category?
```

```
Trading the following items:
  No items to trade.
To receive:
  coin x2
 0: Yes
 1: No
Do you wish to continue? 0

Ready.
Trade complete
activate(bag0)
 0: Open
 1: Activate item
 2: Combine items
 3: Cancel
Which action to perform? 0
Performing action... Open
 0: CATEGORY_DEFAULT
 1: received
Which category? 1
   coin x4

Ready.
```

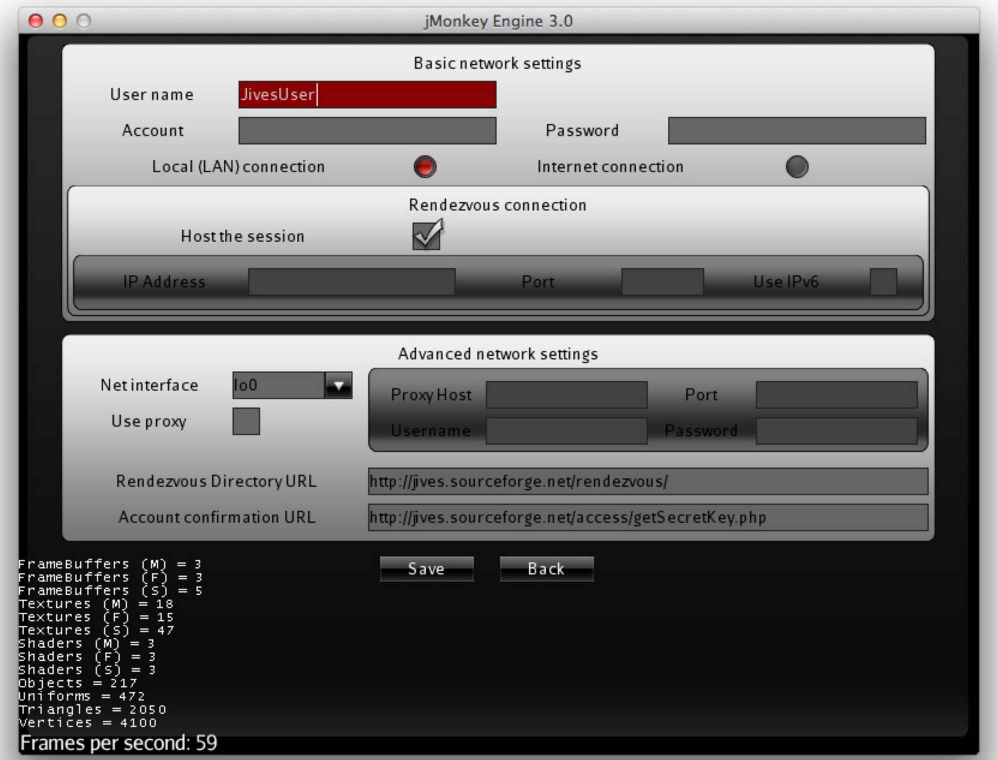
Figure 6.21: Trade between two Playing Characters

By using the *saveState()* and the *loadState()* commands, the user has the possibility to save a snapshot of the current JIVES application and then reload it in a second moment. The argument to be passed to the commands is a string included in double quotes which represents the name of the snapshot to save or load. The snapshot will be saved in the sub-folder ”states” of the folder called ”Roaming\jives” in the ”AppData” directory if the running OS is a Windows system; in the folder *$userHome/.jives/states* if the application is executed in MAC OS X Environment or under a Linux OS. In order to correctly load a save file, the user has first to load the script and then run the *loadState()* command by passing the name of the save enclosed in double quotes.

6.2.2 jME User's Manual

In order to show the possibility to develop and run full 3D Virtual Environments, a jME [31] implementor has been developed. The jME [31] Implementor is exactly a 3D restatement of the Shell Implementor. In order to launch a JIVES Virtual Environment application, the procedure is the same as in the case of a JIVES application built using the Shell Implementor: JIVES essentially consists of a runnable *jar* file, executable by launching the *java -jar testJME.jar* command in a terminal or a shell. When launching JIVES a directory ".jives" ("Roaming\jives" in Windows) is created and includes a folder called "config" which in turn has a sub-folder "jxse" in which is located the *networkConfiguration.xml* file. In the case of a jME [31] based JIVES application, there is no need to modify manually the network configuration file, because JIVES provides an easy to use Graphical User Interface to manage the network configuration settings. Once JIVES has been started, the jME [31] Display Settings windows will appear. The user can set the resolution, the colour depth, the Anti-aliasing parameter, and the Full-screen option. Once set and confirmed the Display Settings parameters, a totally black screen appears: by typing the "Esc" key a short menu is shown or hidden, through which the user may decide to configure the network settings or quit the application. By choosing the "Network settings" button, the Network Settings Graphical User Interface is visualized, as shown in Figure 6.22.

The network settings are exactly the same already seen in the Shell User's Manual. The user can choose the name of its avatar by filling the *Username* field; start the JIVES application using a LAN connection or a Internet connection by selecting the proper checkbox. In the case of a LAN connection, the user may start the JIVES application acting as a Rendezvous by checking the *Host the session* box, or connecting to a Rendezvous running the same JIVES application in the



User name: JivesUser
Account:
Password:
Local (LAN) connection ●
Internet connection
Host the session ✓
IP Address:
Port:
Use IPv6:
Net interface: lo0
Use proxy:
Proxy Host:
Port:
Username:
Password:
Rendezvous Directory URL: http://jives.sourceforge.net/rendezvous/
Account confirmation URL: http://jives.sourceforge.net/access/getSecretKey.php

```
FrameBuffers (M) = 3
FrameBuffers (F) = 3
FrameBuffers (S) = 5
Textures (M) = 18
Textures (F) = 15
Textures (S) = 47
Shaders (M) = 3
Shaders (F) = 3
Shaders (S) = 3
Objects = 217
Uniforms = 472
Triangles = 2050
Vertices = 4100
Frames per second: 59
```

Figure 6.22: Network Settings Graphical User Interface

same LAN network by deselecting the *Host the session* box and specifying the *IP Address* and *Port* fields with the IPv4 and the port of the Rendezvous to which he wants to connect. It is also possible to check the *Use IPv6* box in order to use an IPv6 connection. Note that JIVES offers the IPv6 connection as an experimental feature. In the case the *Use IPv6* box is selected, obviously the user has to fill the *IP Address* field with the IPv6 address of the Rendezvous to which he wants to connect. In the case the user chooses the Internet connection to run the JIVES application, all the parameters included in the "Rendezvous connection" tab are not modifiable any more, because it is JIVES itself to check the Rendezvous directory on the server and decide to start the user as a Rendezvous or a simple Peer.

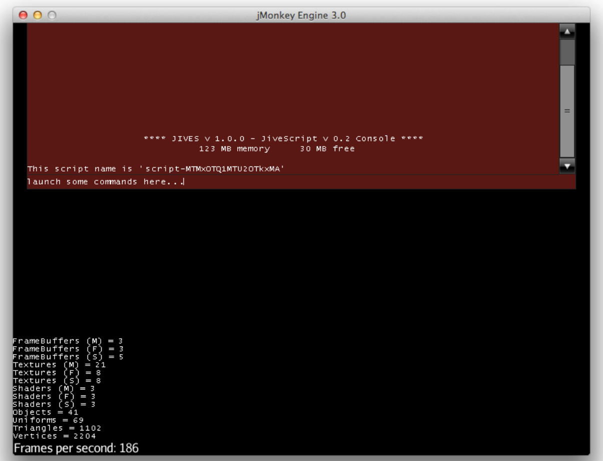
The user has to specify the working network interface by selecting it in the *Net interface* list: all the network interfaces of the machine on which JIVES is running are listed; the user has only to select the right one. Advanced features are represented by the *Rendezvous Directory URL* and the *Account confirmation URL* fields: the first one is related to the URL in which is stored the Rendezvous directory, the second one is related to the web page in which is executed the algorithm to obtain the secret key needed by a Rendezvous to correctly encrypt and decrypt save states. Both the URLS are by default set to web spaces hosted by SourceForge [24], but the developer of a JIVES application may host the Rendezvous directory and the "secret key" algorithm using a server of his choice. In this case the developer has to include the correct URLs inside the *networkConfiguration.default.xml* file, or communicate them to all the users.

The JIVES Network Configuration Settings Graphical User Interface lets the user manage also a connection that works under a Proxy. In this case the user has to select the *Use proxy* box and then specify the URL of the Proxy server in the *ProxyHost* field, the Proxy port in the *Port* field, the credentials in the *Username* and *Password* fields. All the fields related to the Proxy connection are the ones included in the ”Advanced network settings” tab.

JIVES offers also an experimental feature: the possibility to create an account for each user through which it is possible to save and load the application state related to the user’s avatar. The Network Settings Graphical User Interface gives the possibility to specify the credentials of the account by filling the *Account* and *Password* fields located just below the *Username* field.

Once the user has modified in the right way all the network parameters that interested him, the new network configuration can be saved by clicking the ”Save” button; while by clicking the ”Back” button JIVES returns to the previous black screen without modifying any network parameter.

In order to load a JiveScript Virtual Environment script, the jME [31] Implementor provides a full functional console called *JiveScript Console* through which it is possible to launch all the commands previously seen for the Shell Demo, unless some commands specific for the Shell Implementor. In order to visualize the JiveScript Console, the user has to type the ”F1” key of the keyboard. On a MAC keyboard the JiveScript Console appears by using the ”Fn+F1” key combination. The JiveScript Console is shown in Figure 6.23.



```
        **** JIVES v 1.0.0 - JiveScript v 0.2 Console ****
                   123 MB memory      30 MB free

This script name is 'script-MTMxOTQ1MTU2OTkxMA'
launch some commands here...
```

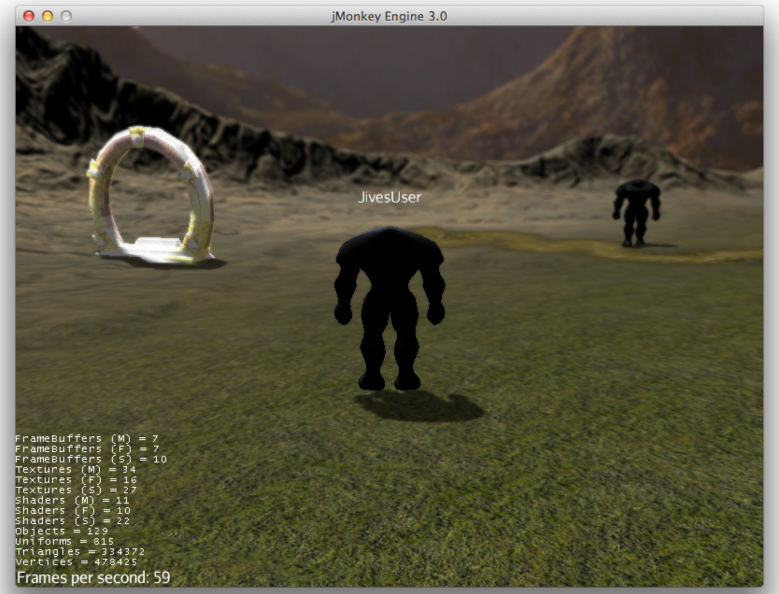
```
FrameBuffers (M) = 3
FrameBuffers (F) = 3
FrameBuffers (S) = 5
Textures (M) = 21
Textures (F) = 8
Textures (S) = 8
Shaders (M) = 3
Shaders (F) = 3
Shaders (S) = 3
Objects = 41
Uniforms = 69
Triangles = 1102
Vertices = 2204
Frames per second: 186
```

Figure 6.23: JIVES JiveScript Console

The list of commands can be displayed by typing the *help()* command in the JiveScript Console.

Sample jME JiveScript

A jME [31] Demo has been developed in order to show the proper functioning of a JIVES 3D Virtual Environment application built by using the jME [31] Implementor. Once JIVES has been started, the user has to open the JiveScript Console by typing the "F1" key on the keyboard and launch the *load()* command passing the protocol (*file:/*) and the absolute path in which the JiveScript is located. The right JiveScript to be loaded is the *basic-demo-jme.jives* file. When the application has been loaded, the user's avatar will be in the opening scene, as shown in Figure 6.24. The statistic nodes that appear in the left bottom corner of the application window can be hidden by typing the "F11" key of the keyboard.



```
FrameBuffers (M) = 7
FrameBuffers (F) = 7
FrameBuffers (S) = 10
Textures (M) = 34
Textures (F) = 16
Textures (S) = 27
Shaders (M) = 11
Shaders (F) = 10
Shaders (S) = 22
Objects = 129
Uniforms = 815
Triangles = 334372
Vertices = 478425
Frames per second: 59
```

Figure 6.24: JiveScript jME Demo Starting Scene

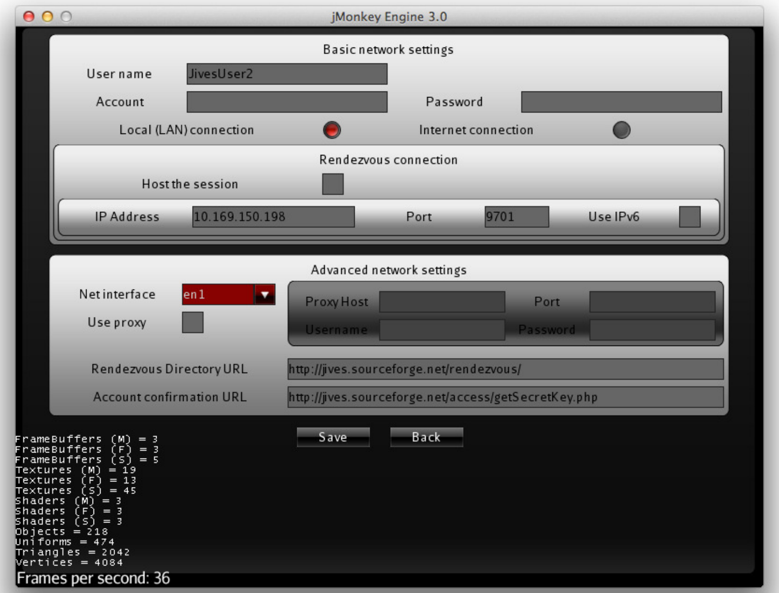
The jME [31] Implementor provides all the operations available in the Shell Implementor. The user can move his avatar by using the "WASD" keys of the keyboard, as typically in First Person Games, and interact with the elements in the scene by using the right mouse button. The camera can be rotated around the avatar by left-clicking the mouse and dragging it in the desired direction, or by using the arrow keys of the keyboard. The user can also zoom in or out by using the middle mouse button. If an other user joins the same JIVES Virtual Environment application, his avatar will appear in the initial scene.

The procedure to join a running JIVES application is similar to the one already seen for the Shell Implementor, but in this case the user has a comfortable Graphical User Interface to set up correctly the network settings. From the point of view of the joiner user, once JIVES has been started, the JIVES Network Configuration Settings have to be accessed by typing the "Esc" key of the keyboard and selecting the "Network settings" button. If the JIVES Virtual Environment application that the second user wants to join is running on a LAN connection, the *Local (LAN) connection* button must be selected while the *Host the session* check-box deselected, and the *IP Address* and *Port* fields must be correctly filled with the IP address and the port of the Rendezvous user.

If the JIVES Virtual Environment application is running using a Internet connection, it is only necessary to select the *Internet connection* button: JIVES itself automatically retrieves from the Rendezvous directory the informations related to the user who acts as a Rendezvous in order to let the second user to join the application. In Figure 6.25 are shown the settings of a user who joins a JIVES application which works using a LAN connection.

Once the second user has joined the application, immediately its avatar appears in the initial scene, as shown in Figure 6.26.

The user can interact both with Non-Playing Characters and Playing Characters and with the Artefacts of the scene: by simply clicking the right mouse button on the element with which the user wants to interact, a list of available actions is shown. Figure 6.27 represents an interaction with a Non-Playing Character.



jMonkey Engine 3.0

Basic network settings

User name: JivesUser2

Account:

Password:

Local (LAN) connection ●

Internet connection

Rendezvous connection

Host the session

IP Address: 10.169.150.198

Port: 9701

Use IPv6

Advanced network settings

Net interface: en1

Use proxy

Proxy Host:

Port:

Username:

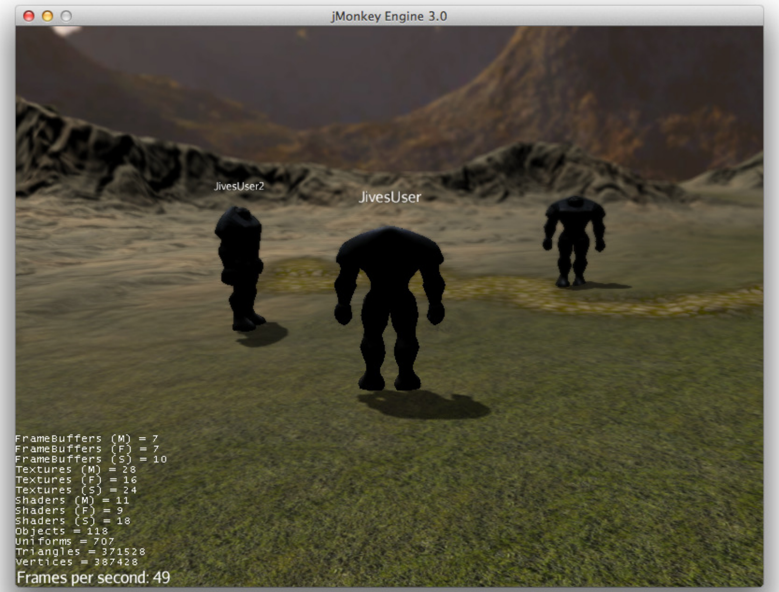
Password:

Rendezvous Directory URL: http://jives.sourceforge.net/rendezvous/

Account confirmation URL: http://jives.sourceforge.net/access/getSecretKey.php

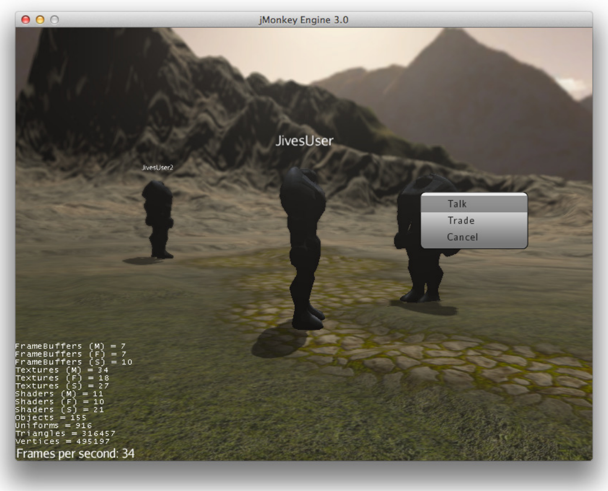
```
FrameBuffers (M) = 3
FrameBuffers (F) = 3
FrameBuffers (S) = 5
Textures (M) = 19
Textures (F) = 13
Textures (S) = 45
Shaders (M) = 3
Shaders (F) = 3
Shaders (S) = 3
Objects = 218
Uniforms = 474
Triangles = 2042
Vertices = 4084
Frames per second: 36
```

Figure 6.25: JIVES Peer jME Network Configuration Settings



```
FrameBuffers (M) = 7
FrameBuffers (F) = 7
FrameBuffers (S) = 10
Textures (M) = 28
Textures (F) = 16
Textures (S) = 24
Shaders (M) = 11
Shaders (F) = 9
Shaders (S) = 18
Objects = 118
Uniforms = 707
Triangles = 371528
Vertices = 387428
Frames per second: 49
```

Figure 6.26: A second user joins the Scene



```
FrameBuffers (M) = 7
FrameBuffers (F) = 7
FrameBuffers (S) = 10
Textures (M) = 34
Textures (F) = 18
Textures (S) = 27
Shaders (M) = 11
Shaders (F) = 10
Shaders (S) = 21
Objects = 155
Uniforms = 916
Triangles = 316457
Vertices = 495197
Frames per second: 34
```

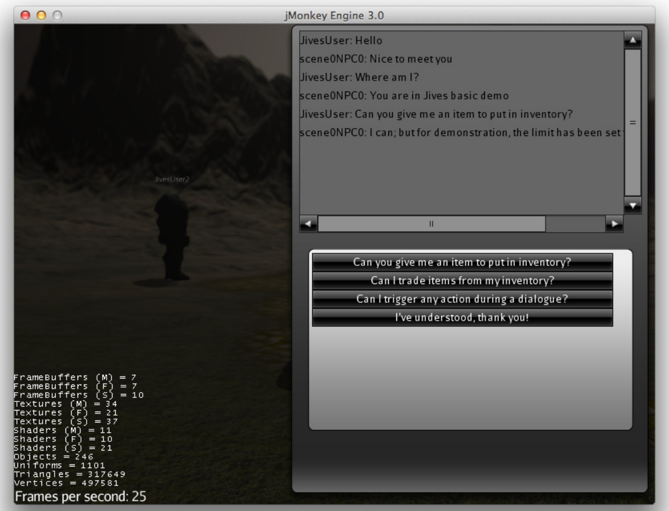
Figure 6.27: Interacting with a NPC

By selecting the "Talk" action when interacting with a NPC it is possible to access to the predefined dialogue related to the given NPC. A dialogue with a NPC is proposed in Figure 6.28.

It is also possible to chat with Playing Characters by activating the PC with the right mouse button and selecting the "Chat" action from the actions list proposed to the user. A chat between two Playing Characters is shown in Figure 6.29.

The user can also chat with an other Playing Character by following a different procedure: he has to activate his avatar by right-clicking on it and selecting the "Chat" action. In the chat window an "Invite" button permits to the user to invite one or more Playing Characters currently present in the same scene. The invite menu is proposed in Figure 6.30.

The user can interact also with the so-called Artefacts, the fixed elements of the scene. In the JiveScript jME [31] Demo the starting scene has only one Artefact, a portal that lets the user move from the current scene to the next one. The portal has to be activated by right-clicking the mouse and then select the "Enter" action. Immediately the user's avatar will be "teleported" in the next scene. Figure 6.31 shows the activation of an Artefact of the scene.

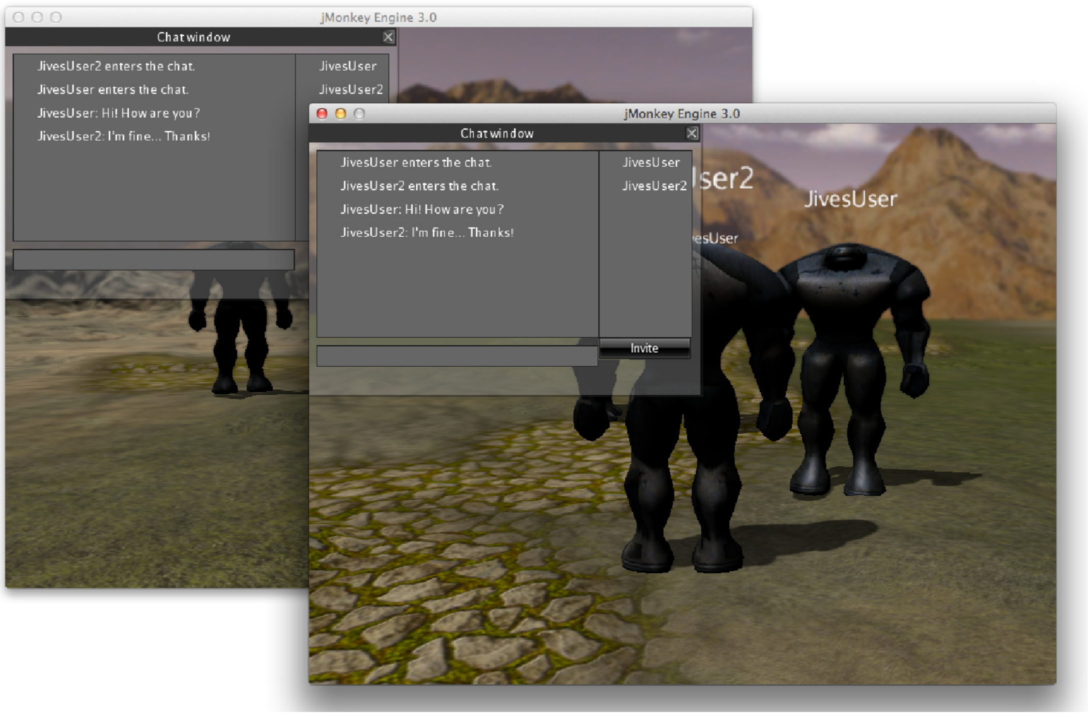

jMonkey Engine 3.0
JivesUser2

JivesUser: Hello
scene0NPC0: Nice to meet you
JivesUser: Where am I?
scene0NPC0: You are in Jives basic demo
JivesUser: Can you give me an item to put in inventory?
scene0NPC0: I can; but for demonstration, the limit has been set

Can you give me an item to put in inventory?
Can I trade items from my inventory?
Can I trigger any action during a dialogue?
I've understood, thank you!

```
FrameBuffers (M) = 7
FrameBuffers (F) = 7
FrameBuffers (S) = 10
Textures (M) = 34
Textures (F) = 21
Textures (S) = 37
Shaders (M) = 11
Shaders (F) = 10
Shaders (S) = 21
Objects = 246
Uniforms = 1101
Triangles = 317649
Vertices = 497581
Frames per second: 25
```

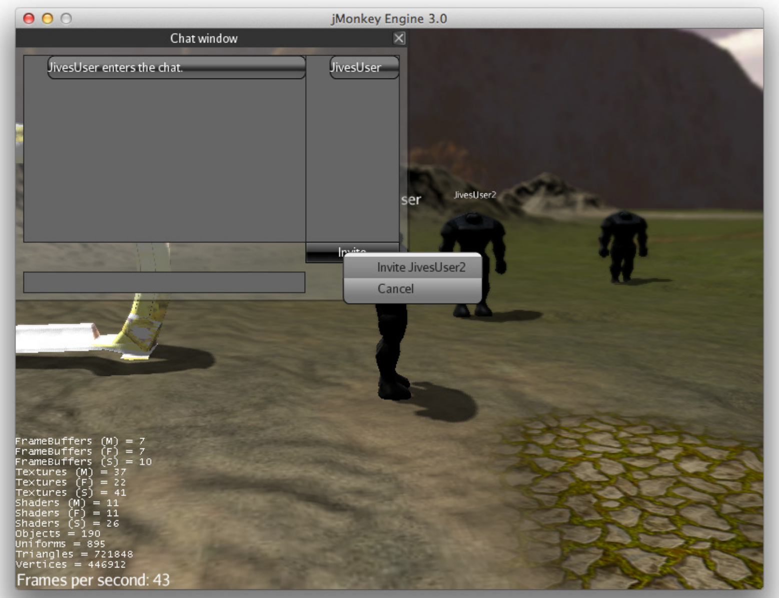
Figure 6.28: Dialogue with a NPC



JivesUser2 enters the chat.
JivesUser enters the chat.
JivesUser: Hi! How are you?
JivesUser2: I'm fine... Thanks!

JivesUser enters the chat.
JivesUser2 enters the chat.
JivesUser: Hi! How are you?
JivesUser2: I'm fine... Thanks!

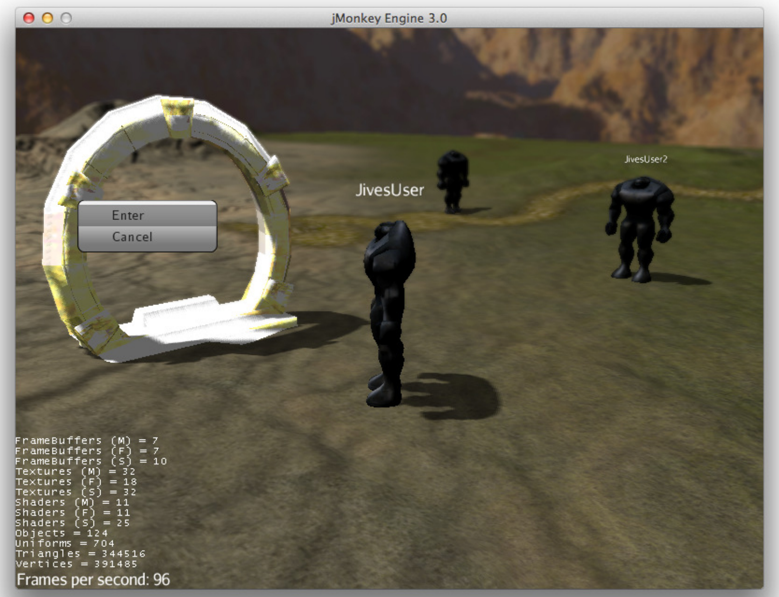
Figure 6.29: Chatting with a PC



JivesUser enters the chat.

```
FrameBuffers (M) = 7
FrameBuffers (F) = 7
FrameBuffers (S) = 10
Textures (M) = 37
Textures (F) = 22
Textures (S) = 41
Shaders (M) = 11
Shaders (F) = 11
Shaders (S) = 26
Objects = 190
Uniforms = 895
Triangles = 721848
Vertices = 446912
Frames per second: 43
```

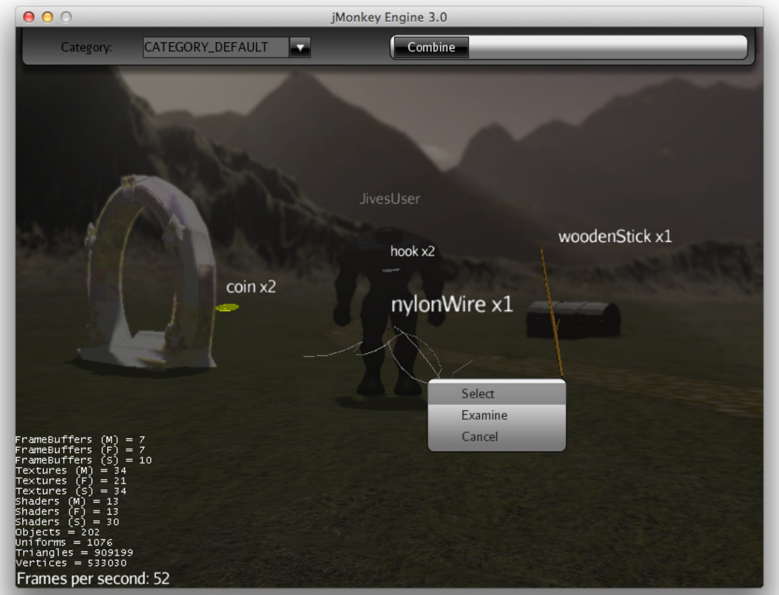
Figure 6.30: Inviting a PC



```
FrameBuffers (M) = 7
FrameBuffers (F) = 7
FrameBuffers (S) = 10
Textures (M) = 32
Textures (F) = 18
Textures (S) = 32
Shaders (M) = 11
Shaders (F) = 11
Shaders (S) = 25
Objects = 124
Uniforms = 704
Triangles = 344516
Vertices = 391485
Frames per second: 96
```

Figure 6.31: Activating the Artifact

Being JIVES an Inventory Based Virtual Environment system, the user has the possibility to collect items from the scenes and manage them inside his inventory. In order to open the inventory, the user's avatar must be activated by clicking the character with the right mouse button and selecting the "Open inventory" action. The Item Inventory Management Graphical User Interface will appear, as shown in Figure 6.32. The user has the possibility to scroll through the various items, examining or selecting them.



```
FrameBuffers (M) = 7
FrameBuffers (F) = 7
FrameBuffers (S) = 10
Textures (M) = 34
Textures (F) = 21
Textures (S) = 34
Shaders (M) = 13
Shaders (F) = 13
Shaders (S) = 30
Objects = 202
Uniforms = 1076
Triangles = 909199
Vertices = 533030
Frames per second: 52
```

Figure 6.32: JIVES Item Inventory Graphical User Interface

If two items are selected, JIVES offers the possibility to combine them in an original and significant way: each item has been previously defined by the developer of the JIVES application as an item characterized by the presence of one or more HotSpots, key points through which the item is available to be combined. The possible combinations of the jME [31] Demo have been defined in the JiveScript jME [31] Demo. In order to access the Combination Screen, the user has to select two items by right-clicking them and choosing the "Select" action. JIVES will ask the user in which quantity he wants to select the item by showing a slider menu. The selected items are shown in the left bottom corner of the window, as depicted in Figure 6.33.

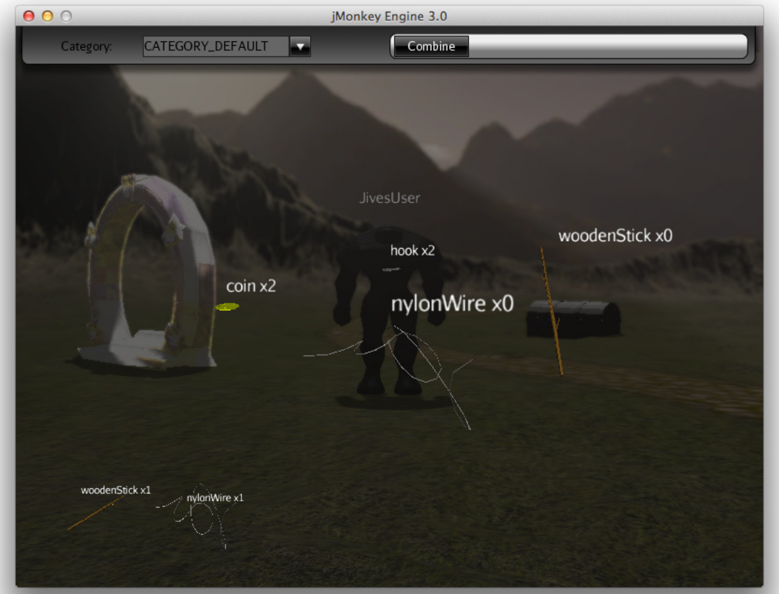
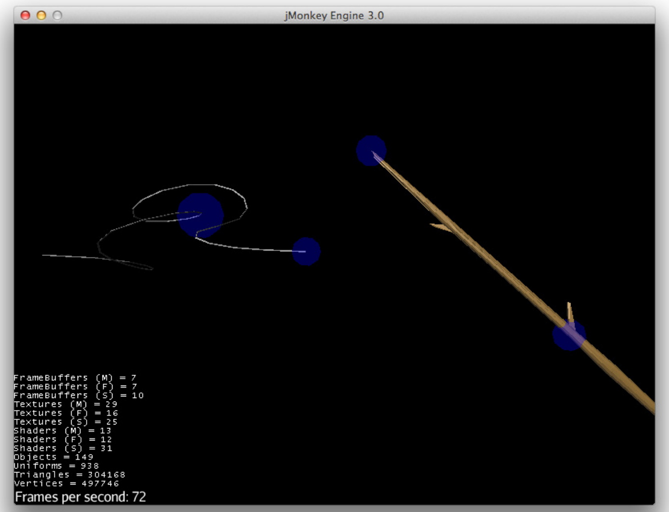


Figure 6.33: Selecting the Items from the Inventory

Once the user has selected the items he wants to combine, the combination can be started by clicking the "Combine" button at the top of the window. The Combination Screen will appear: the user can rotate and translate each item and try to combine the items by joining their HotSpots. When two HotSpots are correctly aligned, the user can accost them by using the "space-bar" button of the keyboard or the middle button of the mouse. In order to make the Combination System more user friendly, a HotSpot Combination Helper is provided. The user can directly select the desired HotSpots by clicking them and they will be automatically aligned, ready to be combined. The user has only to conclude the combination by using the space bar or the middle mouse button. The combination will succeed only when the right HotSpots have been merged. In Figure 6.34 is represented the Combination Screen; while in Figure 6.35 the result of a successful combination.



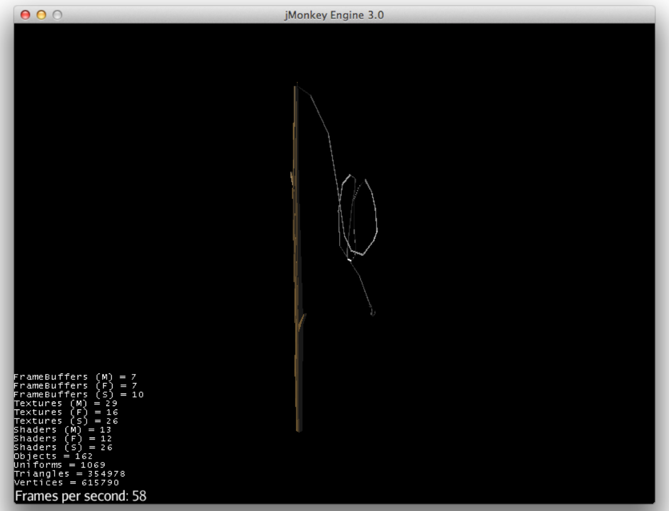
```
FrameBuffers (M) = 7
FrameBuffers (F) = 7
FrameBuffers (S) = 10
Textures (M) = 29
Textures (F) = 16
Textures (S) = 25
Shaders (M) = 13
Shaders (F) = 12
Shaders (S) = 31
Objects = 149
Uniforms = 938
Triangles = 304168
Vertices = 497746
Frames per second: 72
```

Figure 6.34: JIVES Combination Screen

Trading is possible both with Non-Playing Characters and Playing Characters. Trading with Non-Playing Characters is predefined by the JIVES developer; while trading between Playing Characters is totally at discretion of the users: it can consist in an exchange of items, or even in a ”gift” by a user to another one. In both cases trading is activable by right-clicking the character of interest and selecting the ”Trade” action. When the two traders are Playing Characters, once a user has requested a trade to another user, the trade request will appear to the latter one, as shown in Figure 6.36.

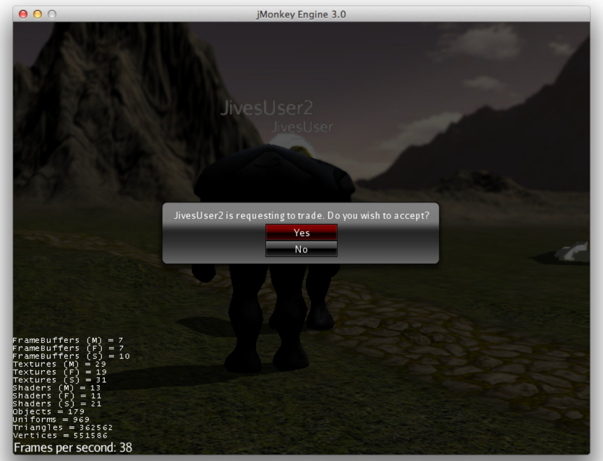
If the user who received the request accepts the trade, a notification will appear to the first trader. Figure 6.37 represents the moment in which the notification is received.

After the trade has been accepted, it is the turn to select the items to exchange: in order to do that each user has to open his inventory and select one or more items, or even no one. When the items have been selected the user has to confirm



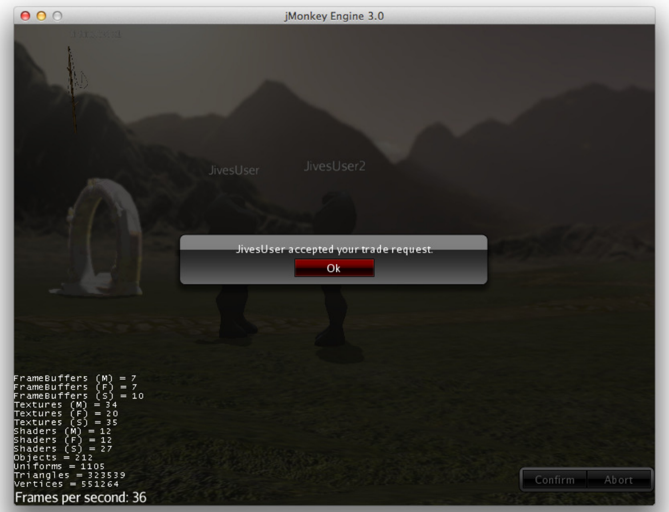
```
FrameBuffers (M) = 7
FrameBuffers (F) = 7
FrameBuffers (S) = 10
Textures (M) = 29
Textures (F) = 16
Textures (S) = 26
Shaders (M) = 13
Shaders (F) = 12
Shaders (S) = 26
Objects = 162
Uniforms = 1069
Triangles = 354978
Vertices = 615790
Frames per second: 58
```

Figure 6.35: A successful combination



```
FrameBuffers (M) = 7
FrameBuffers (F) = 7
FrameBuffers (S) = 10
Textures (M) = 29
Textures (F) = 19
Textures (S) = 31
Shaders (M) = 13
Shaders (F) = 11
Shaders (S) = 21
Objects = 179
Uniforms = 969
Triangles = 362562
Vertices = 551586
Frames per second: 38
```

Figure 6.36: A Trade Request



JivesUser accepted your trade request.

```
FrameBuffers (M) = 7
FrameBuffers (F) = 7
FrameBuffers (S) = 10
Textures (M) = 34
Textures (F) = 20
Textures (S) = 35
Shaders (M) = 12
Shaders (F) = 12
Shaders (S) = 27
Objects = 212
Uniforms = 1105
Triangles = 323539
Vertices = 551264
Frames per second: 36
```

Figure 6.37: The Notification that Trade Request has been accepted

the offer by clicking the "Confirm" button, as depicted in Figure 6.38.

When both the users have confirmed the trade, the items to be exchanged are visualized in the application screen: the items that the user sends appear in the left bottom corner of the window, while the ones that the user receives in the left top corner of the window. If the users are still willing to conclude the trade, it is sufficient that both click the "Trade" button located in the right bottom corner of the application window, as represented in Figure 6.39.

Finally, trade is complete and a notification will appear to both users. Figure 6.40 shows the notification of trade complete.

In the example shown in the screen-shots, it is clearly visible that the trade has been successfully accomplished: the first user offers to the second one a fishing rod, while the second user wants to exchange two hooks. At the end of the trade, as shown in Figure 6.41, the second user has the fishing rod in his inventory and has no more hooks, while the first one has received the two hooks and no longer has the fishing rod.

Trading with Non-Playing Characters can be performed through a different

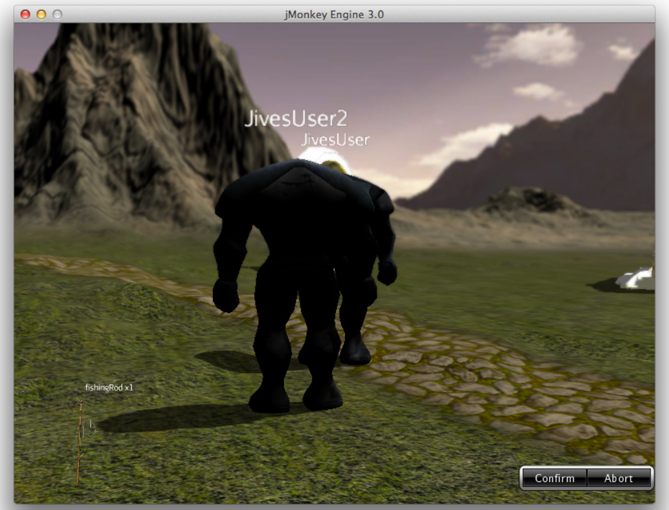


Figure 6.38: Trading the selected Item

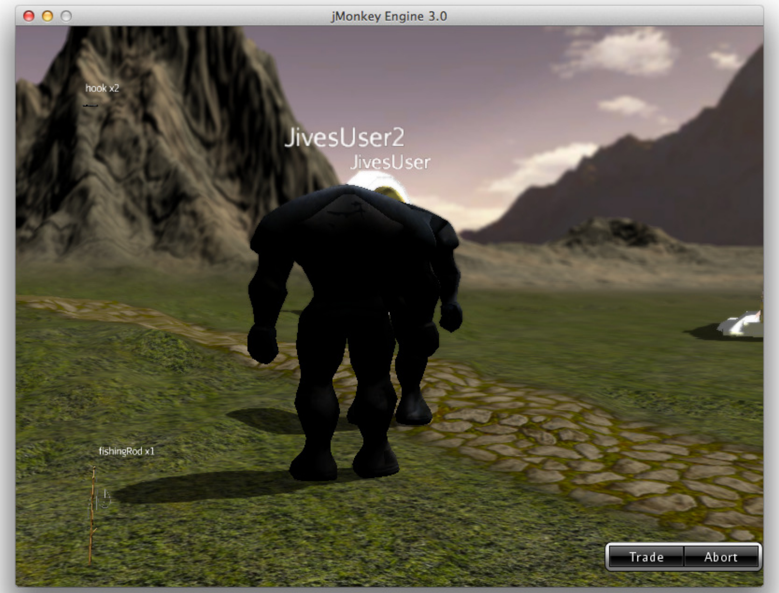


Figure 6.39: Confirming the Trade

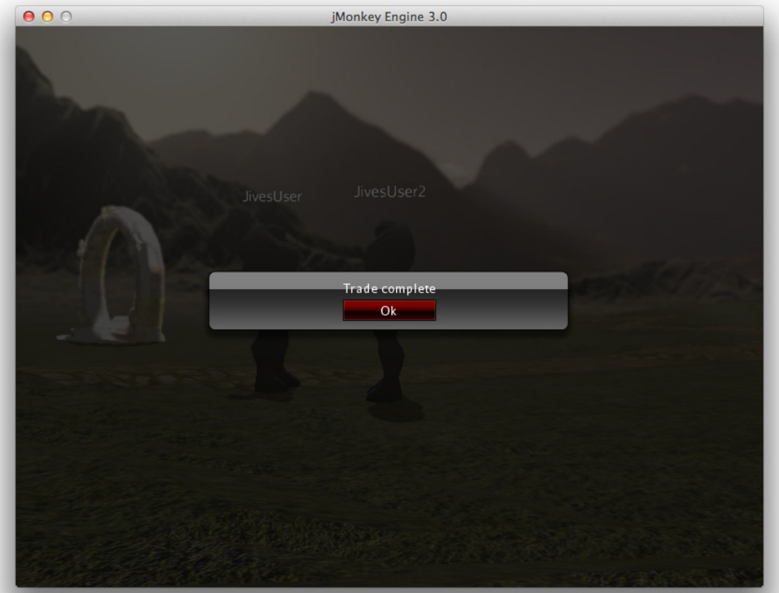
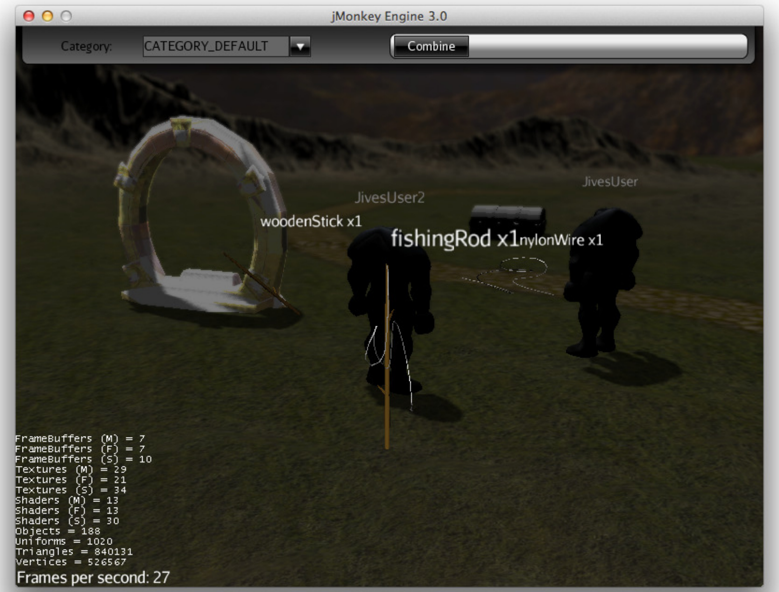


Figure 6.40: Trade is complete



```
FrameBuffers (M) = 7
FrameBuffers (F) = 7
FrameBuffers (S) = 10
Textures (M) = 29
Textures (F) = 21
Textures (S) = 34
Shaders (M) = 13
Shaders (F) = 13
Shaders (S) = 30
Objects = 188
Uniforms = 1020
Triangles = 840131
Vertices = 526567
Frames per second: 27
```

Figure 6.41: The user has received the new Item

procedure with respect to the case in which the traders are two Playing Characters. The NPC has to be activated by selecting the "Trade" Action. The screen proposed in Figure 6.42 will be shown to the user.

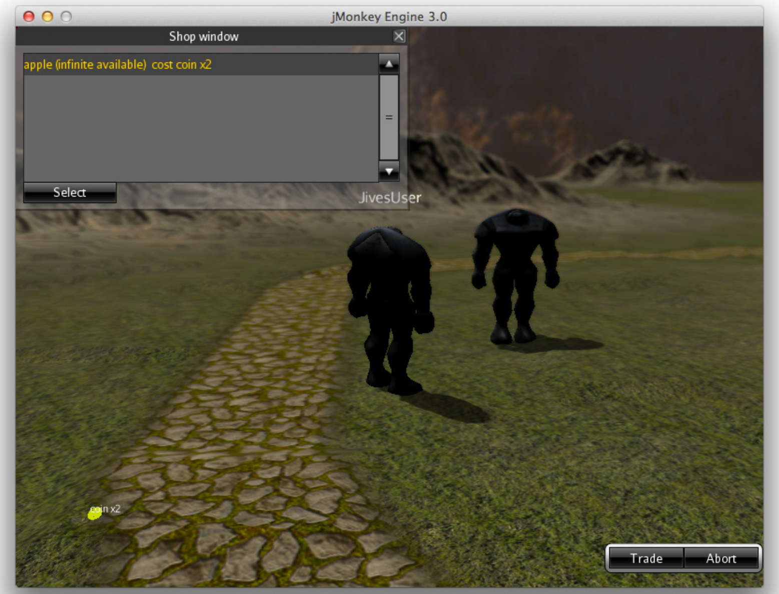


Figure 6.42: Trading with a NPC

At the top left of the window there is the so-called "Shop window" which informs the user the item(s) will receive and the item(s) he has to give in order to accomplish successfully the trade. The needed items are to be selected in the Inventory Screen, by right-clicking the user's avatar and selecting the "Open inventory" action. Once the items have been selected, they will appear in the bottom left corner of the window. In the "Shop window" the desired trade must be selected using the "Select" button: if the needed items correspond to the ones selected by the user, the trade can be performed by clicking the "Trade" button. A box informing the user that the trade has been successfully completed will appear. At the contrary, if something is not set in the way that the trade requires a box will appear communicating that the trade has been aborted.

The jME [31] Implementor offers in the same way as the Shell Implementor the possibility to save or load a snapshot of current state of the running JIVES application. The user has to open the JiveScript Console by typing the ”F1” key of the keyboard and then launch the *saveState()* command in order to save the snapshot or the *loadState()* command in order to load a snapshot previously saved. The argument to be passed to the commands is the name of the snapshot and it must be included into double quotes. The locations where the snapshot is saved are exactly the same ones already described in the Shell User’s Manual: a folder called ”Roaming\jives\states” in the ”AppData” directory if the running OS is a Windows system; *$userHome/.jives/states* if the application is executed in MAC OS X Environment or under a Linux OS. All the save files are stored in the sub-folder ”states”.

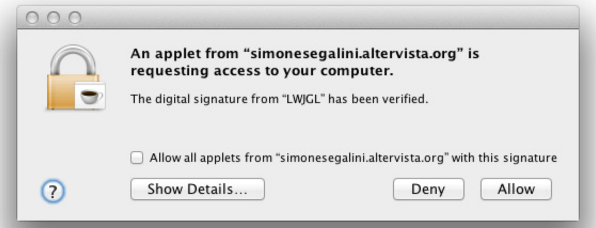
6.2.3 Java Applet User’s Manual

A JIVES Virtual Environment application can be deployed also as a Java [51] Applet that runs inside a web browser. In order to test this feature, the jME [31] Demo has been published at the following URL: `http://simonesegalini.altervista.org/jives/test/test.html`. An essential prerequisite to correctly run the JIVES Applet is to grant the right permissions to the application in order to access the user’s computer. This is possible by providing a trusted certificate with which the applet is signed. For testing purposes, however it’s sufficient to place the *.java.policy* file in the right location: in a Windows OS this file has to be located in *C:\Documents and Settings\$user\.java.policy* or *C:\Users\$user\.java.policy* depending on the specific Windows version; in a MAC OS X Environment in */Users/$user/.java.policy*; under a Linux system in the ”home” directory */home/$user/.java.policy*. The *.java.policy* file is not provided to the user: he has to create it manually, open it with any text editor and insert the content shown in Code Snippet 6.5.

Code Snippet 6.5: The Policy File

```
// Do what you will. Totally permissive policy file.
grant {
permission java.security.AllPermission;
};
```

When accessing the Launcher Page for the JIVES Applet, a message box, as shown in Figure 6.43, displays the request that the JIVES Applet wants to access to the user's computer. This happens because the applet is signed with a digital certificate. This certificate claims that the applet comes from the party named within and contains the digital signature of a certificate authority. In order to correctly run the applet the user must accept the certificate request.


An applet from "simonesegalini.altervista.org" is requesting access to your computer.

The digital signature from "LWJGL" has been verified.

☐ Allow all applets from "simonesegalini.altervista.org" with this signature

Figure 6.43: The Certificate Request

Because the *JMEImplementor* is quite memory consuming to run as a Java Applet, the user might have to increase the Java [51] Heap of its Java [51] Virtual Machine, at least augmenting it to 300 Mb. This is possible by defining in the JVM Settings of the user's OS the parameter *-Xmx300m*. See `http://java.sun.com/performance/reference/whitepapers/tuning.html#section4.1.2` for further informations about increasing the Java Heap size. The first time that the user runs the JivesApplet, it takes a long time to start because the Java [51] Web Start has to download all the necessary packets from the server. These packets are then stored in the Java [51] Web Start cache: this permits to load the JivesApplet in a much shorter time.

6.3 Sample JiveScripts

Both the JiveScript Shell Demo and the JiveScript jME [31] Demo at the beginning of the scripts make use of two important directives: the directive *__uses* and the directive *__name*. As shown in Code Snippet 6.6, the *__uses* directives will be checked by the system to ensure that all the Implementors needed by the JiveScript application are available. In the JiveScript Shell Demo as argument of the directive is passed the Shell Implementor, while in the JiveScript jME [31] Demo is passed the jME [31] Implementor. The *__name* directive is used to define the name of the JIVES application.

Code Snippet 6.6: *__uses* and *__name* directives

```
// JIVESCRIPT_VERSION = 0.2
reset();

__uses(org.jives.implementors.engine.shell);
__name("Jives Basic Demo");

echo("");
echo("");
echo("--------------------------------------------------------------------");
echo("  Welcome to Jives Basic Demo");
echo("--------------------------------------------------------------------");
echo("This is a textual adventure engine that uses "
        + "Jives Shell implementor and JiveScript.");
echo("Say \"help()\" for a list of available commands.");

...

// Prevent further scripting
__scripting(false);
```

At the end of the script, the *__scripting* directive is defined. Usually, when the script is in its final version, this directive is set to false, as in the JiveScript Demo. In this case the script can be only loaded and not modified any more. Setting the directive *__scripting* to true allows the JIVES developer to enable a "sandbox" modality through which he has the possibility to freely experience some changes in the JIVES application he's developing.

The Setup section of the JiveScripts define the listeners and the item combinations provided by the application. The JIVES framework manages the events through an event-based system: each event generated inside the application has a condition and an action. If the condition is verified, the action is executed. Through the use of listeners, or better called event listeners, there is the possibility to notify and manage the event, executing or inhibiting the associated action. As shown in Code Snippet 6.7, in the JiveScript Demos the user's avatar can receive from the NPC of the first scene at most two *coins*; in the case the user asks for a third *coin* the listener inhibits the action associated to the given event.

Code Snippet 6.7: Event listener

```
// Define a listener
inventoryListener = new org.jives.events.JivesEventListenerIntf({
  handleEvent: function (jivesEvent) {
        action = jivesEvent.getAction();

        ...

        if (action instanceof org.jives.actions.BagItemAction) {
            if(bag0.getQuantity("coin") < 2) {
                gui.notification("[Listener] Received "
                        + action.getItem().getId() + ".");
                return false;
            } else {
                gui.notification("[Listener] You already have enough "
                + "coins.");
                return true;
            }
        }

        ...

        return false;
    },
  getId: function () { return "inventoryListener"; }
});
Jives.registerEventListener(inventoryListener);
```

The items combinations in the JiveScript Shell Demo are defined as shown in Code Snippet 6.8: each item is characterized by an ID, the category to which it belongs, its HotSpots and its model. For each HotSpot, if a combination is defined, the item that will be created as a result of a correct combination is reported. In the model definition is specified the description of the given item; description that appears to the user when he selects the ”Examine” action from the Item Inventory Screen.

Code Snippet 6.8: Shell Items combinations

```
// Define bag0
bag0 = makeBag({
    bag0 : [

        ...

        {
            id: "woodenStick",
            categories: [ "found" ],
            combines: [
                [ new org.jives.implementors.engine.shell
                        .HotspotRenderer("bottom"), null ],
                [ new org.jives.implementors.engine.shell
                        .HotspotRenderer("tip"), "stickWithWire" ]
            ],
            model: new org.jives.implementors.engine.shell.BagItemRenderer(
                "A strong and flexible wooden stick"
            ),
            commonActionsRenderer: gui
        },
        {
            id: "nylonWire",
            categories: [ "found" ],
            combines: [
                [ new org.jives.implementors.engine.shell
                        .HotspotRenderer("edge"), "stickWithWire" ],
                [ new org.jives.implementors.engine.shell
                        .HotspotRenderer("middle"), null ]
            ],
            model: new org.jives.implementors.engine.shell.BagItemRenderer(
                "A piece of nylon wire"
            ),
            commonActionsRenderer: gui
        },

        ...

    ]
}, bagModel, bagModel);
```

In the JiveScript jME [31] Demo the only differences are that for each HotSpot are defined in space using 3D coordinates for positioning and a radius for sensibility, and the item model refers to the mesh through which the item is visualized in a 3D Virtual Environment. Code Snippet 6.9 shows how the items are defined in the jME [31] Demo script.

Code Snippet 6.9: jME Items combinations

```
// Define bag0
bag0 = makeBag({
    bag0 : [

   ...

    {
        id: "woodenStick",
        categories: [ "found" ],
        combines: [
            [
               new org.jives.implementors.engine.jme.HotspotRenderer(
                  com.jme3.math.Vector3f(-3.42, -3.08, 0.19), 1
               ), null
            ],
            [
               new org.jives.implementors.engine.jme.HotspotRenderer(
                  com.jme3.math.Vector3f(9.5, 7.8, -0.03), 1
               ), "stickWithWire"
            ]
        ],
        model: new org.jives.implementors.engine.jme.BagItemRenderer(
            "Models/woodenStick/woodenStick.mesh.xml", 0.05,
            "A strong and flexible wooden stick"
        ),
        commonActionsRenderer: gui
    },
    {
        id: "nylonWire",
        categories: [ "found" ],
        combines: [
            [
               new org.jives.implementors.engine.jme.HotspotRenderer(
                  com.jme3.math.Vector3f(-10.29, 1, -0.91), 1
               ), "stickWithWire"
            ],
            [
               new org.jives.implementors.engine.jme.HotspotRenderer(
                  com.jme3.math.Vector3f(2.54, 1.25, 6.24), 2
               ), null
            ]
        ],
        model: new org.jives.implementors.engine.jme.BagItemRenderer(
            "Models/nylonWire/nylonWire.mesh.xml", 0.016,
            "A piece of nylon wire"
        ),
        commonActionsRenderer: gui
    },

    ...

 ]
}, bagModel, bagCombiner);
```

Code Snippet 6.10 presents how the scene models and the elements inside the scene are defined. For each element in the scene are bound predefined actions, that are the ones shown to the user when he activates it. In Code Snippet 6.10, Scene 2 is set with the *scene2PortalTo1* Artefact. To that element is bound the *PortalAction* action, that lets the user to move from Scene 2 to the previous one.

Code Snippet 6.10: Scene Model Definition

```
// Define scene models
scene2Model = new org.jives.implementors.engine.shell.SceneRenderState(
    new org.jives.sim.SceneModelIntf({
        destroy: function () { },
        activate: function () {
            echo("\n\nSCENE 2:\n This room has a quiet pond with crystal waters.");
        },
        onActiveNodeAdd: function (jivesActiveNode) {
            if (jivesActiveNode.getModelType()
                .equals(org.jives.network.RemoteActiveNode)) {
                org.jives.implementors.engine.shell.SceneRenderState
                        .setupTradeAction(jivesActiveNode, bag0);
                org.jives.implementors.engine.shell.SceneRenderState
                        .setupRemoteDialogue(jivesActiveNode);
            }
        },
        onActiveNodeRemove: function (jivesActiveNode) { }
    })
);
scene2 = new org.jives.core.JivesScene("Scene 2", scene2Model);

...

// ------------------------------ Scene 2 --------------------------------- //

// Door to scene 1 --------------------------------------------------------
scene2PortalTo1 = new org.jives.core.JivesActiveNode("scene2PortalTo1",
    new org.jives.implementors.engine.shell.ActiveNodeRenderer(
        org.jives.implementors.engine.shell.ActiveNodeRenderer.TYPE_PORTAL,
        "Portal to Scene 1"
    )
);
scene2.addActiveNode(scene2PortalTo1);
scene2PortalTo1.bindAction(
    new org.jives.actions.PortalAction(
        "scene2PortalTo1Action",
        scene2, scene1
    )
);
```

In the initial scene of the JiveScript Demo, there is also a Non-Playing Character defined with a preset dialogue. The whole dialogue is reported before the definition of the NPC and then bound to it by using the *StartDialogueAction* action. The dialogue requests are structured into three main parts: the question, the answer and the actions related to that answer. In Code Snippet 6.11 a part of the dialogue related to the NPC of the first scene is presented.

Code Snippet 6.11: A predefined Dialogue

```
// NPC ------------------------------------------------------------------------
// Dialogue 0
scene0NPC0Dialogue0Root = makeDialogue({
    scene0NPC0Dialogue0Root : [
        {
            question: "Hello",
            answer: "Nice to meet you",
            actions: [
                new org.jives.actions.StartDialogueAction(
                    "scene0NPC0Dialogue0RootReq0Action", "scene0NPC0Dialogue0Intro"
                )
            ]
        }
    ]
}, dialogueModel, gui);

makeDialogue({
    scene0NPC0Dialogue0Intro : [
        {
            question: "Where am I?",
            answer: "You are in Jives basic demo",
            actions: [
                new org.jives.actions.StartDialogueAction(
                   Jives.generateId(), "scene0NPC0Dialogue0Dialogues"
                )
            ]
        },
        {
            question: "Who are you?",
            answer: "I'm a non-playing character. I'm here to demonstrate the "
                + "dialogue system. Ask your questions.",
            actions: [
                new org.jives.actions.StartDialogueAction(
                    Jives.generateId(), "scene0NPC0Dialogue0Dialogues"
                )
            ]
        },
        {
            question: "Have a nice day",
            answer: "Goodbye"
        }
    ]
}, dialogueModel, gui);

makeDialogue({
    scene0NPC0Dialogue0Dialogues : [

        ...

        {
            question: "Can I trigger any action during a dialogue?",
            answer: "Of course; Here, an action is triggered and then
                the dialogue continues",
            actions: [
                new org.jives.core.JivesAction("inDialogueAction",
                        new org.jives.sim.JivesActionModelIntf({
                    execute: function (jivesEvent) {
                        gui.notification("Light blinks and walls tremble
                        for a moment...");
                    },
                    getDescription: function () {  return "In-dialogue action"; },
                    render: function (jivesRenderableIntf) { gui.notification(
                    this.getDescription()); }
                }})),
                new org.jives.actions.StartDialogueAction(
                    Jives.generateId(), "scene0NPC0Dialogue0Dialogues"
                )
            ]
        },

        ...

    ]
}, dialogueModel, gui);

// Bind dialogues to NPC
scene0NPC0 = new org.jives.core.JivesActiveNode("scene0NPC0",
    new org.jives.implementors.engine.shell.ActiveNodeRenderer(
        org.jives.implementors.engine.shell.ActiveNodeRenderer.TYPE_NPC,
        "Guy in the southwest corner"
    )
);
scene0.addActiveNode(scene0NPC0);
scene0NPC0.bindAction(
    new org.jives.actions.StartDialogueAction(
        "scene0NPC0Dialogue0RootAction",
        scene0NPC0Dialogue0Root.getId()
    )
);
```

The JiveScript Demo proposes also a trade between a Playing Character and the Non-Playing Character of the initial scene. The trade is defined in the JiveScript as shown in Code Snippet 6.12, specifying which items the NPC has to receive in order to accomplish the trade, and which items the NPC gives to the user. In the proposed example, the NPC will receive two *coins* in order to give a *apple* to the user.

Code Snippet 6.12: Trade with a NPC

```
// Bind trade to NPC
tradeItem = Jives.get("apple", org.jives.core.JivesBagItem);
catalog = new org.jives.implementors.engine.shell.ShopCatalog();
catalog.addEntry(tradeItem, -1, dialogueItem, 2);
org.jives.implementors.engine.shell.SceneRenderState
        .setupTradeAction(scene0NPC0, bag0, catalog);
```

The whole JiveScript Shell Demo and JiveScript jME [31] Demo are reported in Appendices B and C.

Chapter 7

Performance Metrics and Evaluation

7.1 Performance Metrics

The performance metrics discussed in this section are both quantitative and qualitative. In order to evaluate the quantitative performance metrics an experiment based on the execution of the jME [31] Test Demo has been set up using different hardware machines. The hardwares involved are a MacBookPro 2.3 GHz Intel quadCore i7 with 8 GB 1333 MHz DDR3 and Graphics Card AMD Radeon HD 6750M 1024 MB, an iMac 2.66 GHz Intel dualCore 2 Duo with 4 GB 800 MHz DDR2 SDRAM and Graphics Card ATI Radeon HD 2600 Pro 256 MB, a MacBook 2.26 GHz Intel dualCore 2 Duo with 2 GB 1067 MHz DDR3 and Graphics Card Nvidia Geforce 9400M 256 MB and a Sony Vaio 2.10 GHz Intel dualCore 2 Duo with 4 GB 800 MHz DDR2 SDRAM and Graphics Card Nvidia Geforce 9300M 256 MB. For each hardware the experiment is done on idle processor and averages taken over ten independent runs. In order to test the performance of the network, all the experiments have been reproduced by using a Wireless Local Area Network (WLAN) 802.11n (450 Mb/s) and an Internet ADSL connection with a 8128 kbps downstream connection speed and a 480 kbps upstream connection speed. At start-up, a measure is done on the period of time it takes to initialize JIVES and start the jME [31] Test Demo, in window-mode with a 800x600 pixels resolution, 24 bpp colour depth and anti-aliasing filter disabled.

Table 7.1: Start up experiment results

Number of run	MacBookPro 1	MacBookPro 2	iMac 1	iMac 2	MacBook 1	MacBook 2	Sony Vaio 1	Sony Vaio 2
1	19.30	12.60	21.60	21.40	28.00	28.40	23.10	18.52
2	17.8	11.90	20.90	14.70	27.50	17.10	23.00	17.96
3	17.2	11.10	21.00	16.60	25.80	15.90	22.90	20.88
4	17.0	10.90	20.70	17.10	23.90	15.80	22.50	16.49
5	17.4	11.80	20.70	17.20	24.00	15.80	22.10	17.20
6	16.8	11.20	20.30	14.30	24.5	16.00	22.20	17.38
7	16.4	10.70	20.80	14.70	25.30	15.80	22.80	16.79
8	16.5	11.00	20.40	14.30	24.90	16.20	22.60	18.14
9	16.4	10.90	20.50	14.20	25.70	16.00	22.70	15.05
10	16.6	10.20	20.90	14.00	25.80	16.40	22.30	19.80
Average	17.14	11.23	20.78	15.85	25.54	17.34	22.62	17.82

In Table 7.1 are shown the results obtained from the start-up experiment. Using the MacBookPro, the start-up takes an average of 17.14 seconds to load JIVES and start the Demo, ranging from a minimum of 16.4 seconds and a maximum of 19.3 seconds; the iMac obtains an average of 20.78 seconds with a minimum of 20.3 seconds and a maximum of 21.6 seconds; the MacBook performs the start-up in an average of 25.54 seconds, obtained from a range between 23.9 and 28.0 seconds; on the Sony Vaio the experiment reveals a minimum of 22.1 seconds and a maximum of 23.1 seconds, reaching an average of 22.62 seconds. As a result of this experiment, it has been decided to try to improve the start-up performances by reducing the number of classes that the *MultiInstanceEndpoint* JXSE [54] classloader has to load while starting the network. The start-up experiment has been repeated after the implementation of that optimization, leading to have different performance results. The MacBookPro start-up average time is 11.23 seconds, with a worst time of 12.6 seconds and the best one of 10.2 seconds; the iMac performs the start-up in a range from 14.0 and 21.4 seconds, with an average of 15.85; the MacBook achieves an average of 17.34 seconds, with a minimum of 15.8 and a maximum of 28.4 seconds; the Sony Vaio starts JIVES in a best time of 15.05 and a worst time of 20.88 seconds, obtaining an average value of 17.82 seconds.

The start-up experiment has been done varying the connection type, from LAN connection to Internet connection, but the application is not affected in terms of boot speed: the average times are similar in both cases. The comparative graph depicted in Figure 7.1 visualizes the performances computed from the first experiment with a dashed line and the performances from the second experiment with a solid line.

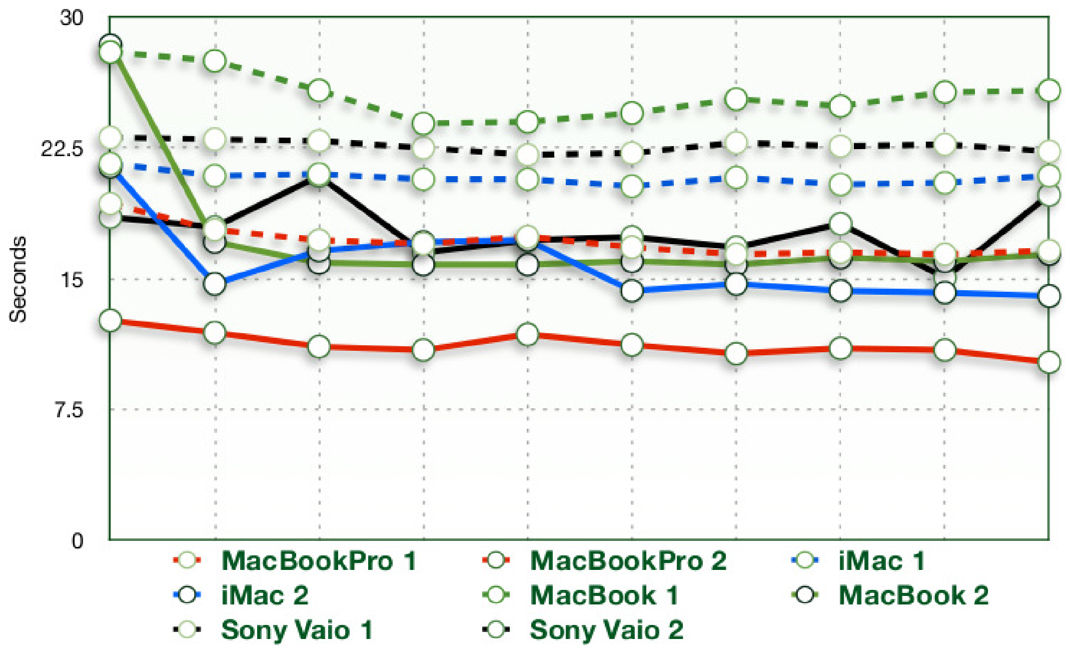


Figure 7.1: Start-up experiment

The optimization introduced by reducing the number of classes loaded by the *MultiInstanceEndpoint* JXSE [54] class-loader has permitted to reduce substantially also the amount of memory consumed by JIVES: starting from an amount of about 600 MB, now the jME [31] Test Demo uses approximately 300 MB, achieving an improvement of 50 % in terms of memory usage.

Due to the fact that JIVES is executable as a Java Applet, experiments have been done in order to test the capacities of the most popular browsers to load and execute the jME [31] Test Demo implemented with the JIVES framework. The Demo has been proved to be successfully executed using Mozilla Firefox, Apple Safari, Google Chrome, the Open Source browser Chromium and Internet Explorer. The only browser that failed to run the Demo, without even starting it, is Opera. The execution of the Demo as a Java Applet, when using a Windows or Linux Environment system, did not reveal any substantial differences in terms of performance metrics while running, except for a slightly bigger network lag when compared to the desktop version, but it's way slower while loading especially if the applet sign has to be verified for every loaded class. This performance decay should not be present when using a trusted certificate. At the contrary, when running the applet using a MAC OS X operating system, were noticed some problems related to the improper handling of input devices, especially the mouse input. In particular, due to the fact that the mouse pointer cannot be captured, there is no longer the possibility to correctly manage the camera by using the mouse. The user however has the possibility to manage the rotation by means of the arrow keys of the keyboard so as to overcome this issue. In the Combination Screen the items can be correctly manipulated by holding the left mouse button on the desired object and rotating it using the arrow keys. Same applies to translation when the right mouse button is pressed.

In order to test the network strength, an experiment has been done by using both a LAN connection and an Internet connection. In the first case, a Rendezvous has started the jME [31] Test Demo; incrementally at one minute interval a new peer (executed on a physical machine) has been set up to connect to the Rendezvous up to obtain a scene shared between a Rendezvous and nine peers. Up to a Rendezvous and five peers, the network has performed well, maintaining a low lag level; while increasing even more the number of connected users results in a degradation of the network that can also lead to have a network lag of more than a second. In particular, the minimum lag observed was 0.6 seconds, while the maximum one 2.1 seconds. In the second case, the execution of the experiment using an Internet connection, the network behaviour is very similar, except of course an increased network lag, depending on the Internet connection speed. Using an ADSL with a 8128 kbps downstream connection speed and a 480 kbps upstream connection speed, the lag increased to a maximum of 3 seconds. In this specific case the distance that the signal had to run through was about 30 km.

In order to test the JXSE network on higher traffic loads, it has been necessary to simulate it, due to the impossibility to provide an adequate number of hardware machines. This kind of testing has been achieved through the *ShellImplementor*. A BASH [21] script has been used to automatically interact with a running instance actually performing like a bot. Through this setup, we were able to perform the network performance analysis. The objective of the experiment was the examination of the worst possible case: as many Peers as possible, connected to a single Rendezvous, simultaneously talking to a single receiver Peer, connected to the same Rendezvous. The experiment took place as follows: all senders sent a message containing the current timestamp once every ten seconds; when the receiver received the message, it measured the time occurred in between delivery.

The whole procedure involves many scripts, listed below; they must be located in the same directory of the *testShell* build; along with them, several output files are created, one for each Peer plus one that contains the logged experiment results. Once the Rendezvous has been started by launching the *testShell* Demo, the network can be instantiated directly, issuing the command *Jives.getNetwork().startNetwork("test", org.jives.implementors.network.jxse.utils.Tools.md5Converter("test"))*. Then, the receiver has to be started, simply running the script reported in Code Snippet 7.1.

Code Snippet 7.1: The Network Test Receiver Script

```
#!/bin/bash
file="./0-output.txt"
log="./0-log.txt"
path="/usr/src/eclipse/jives/out/" # Path of the testShell build

function bot {
  echo "Jives.getNetwork().startNetwork(\"test\",
       org.jives.implementors.network.jxse.utils.Tools.md5Converter(\"test\"))"
  while true; do
    echo "lookAround()"
    sleep 1
    cat $file | while read line; do
      found=$(echo $line | grep ">> peer0")
      if [ -n "$found" ]; then
        recv_time=$(echo $(($(date +%s%N)/1000000)))
        sender=$(echo "$line" | cut -d" " -f4)
        send_time=$(echo "$line" | cut -d" " -f5)
        let DIFF=$recv_time-$send_time
        echo "$sender, " $DIFF >> $log
      fi
    done
    echo "" > $file;

    # Ponder next move
    sleep 0.5
  done
}

cd $path
touch $file
echo "" > $file;
touch $log
echo "" > $log;
bot | java -jar testShell.jar | tee $file
```

At last, the file *0-log.txt* will contain the delivery time of all the messages received and the respective provenance. *0-output.txt* instead is used as storage to parse the output buffer iteratively. The output files are prefixed with a serial ID so that different Peers won't overwrite each others files. Also senders are executed programmatically, once every while, and instructed to send messages to the receiver, named *peer0*. Code Snippet 7.2 shows how they are started, one after the other, calling the *testnet-send.sh* script which actually executes the sender job, as can be seen in Code Snippet 7.3.

Code Snippet 7.2: The Network Test Starter of Sender Scripts

```
path="/usr/src/eclipse/jives/out/" # Path of the testShell build

cd $path

for i in {1..100}
do
  echo "Starting peer nÂ°$i..."
  sh ./testnet-send.sh $i &
  sleep 30
done
```

Code Snippet 7.3: The Network Test Sender Script

```
#!/bin/bash
if [ "$1" == "" ]; then
  echo "Specify index param"
  exit
fi

arg=$1
file="./$arg-output.txt"
path="/usr/src/eclipse/jives/out/"

function bot {
  echo "Jives.getNetwork().startNetwork(\"test\",
        org.jives.implementors.network.jxse.utils.Tools.md5Converter(\"test\"))"
  while true; do
    # Search dest peer
    echo "" > $file;
    echo "lookAround()"
    sleep 1
    found=$(echo $(cat $file) | grep -c "peer0")
    if [ $found == 1 ]; then
      # Dest peer found, talk
      echo "activate(\"peer0\")"
      echo "0"
      time=$(($(date +%s%N)/1000000))
      echo "$arg $time"
    fi

    # Ponder next move
    sleep 9
  done
}

cd $path
touch $file
echo "" > $file
bot | java -jar testShell.jar | tee $file
```

Testing on *localhost*, however, does not give meaningful results because the network propagation is immediate. However, from Linux Kernel 2.6 the *netem* network emulation [28] was introduced as kernel module. Thanks to *netem* [28], it is possible to simulate a network delay of 100ms +/- 10ms having a random distribution issuing the command *tc qdisc add dev lo root handle 1:0 netem delay 100ms 10ms.* To revert emulation just run *tc qdisc del dev lo root.*

As a side note, it must be said that the same procedure can run on a Mac OS X, if opportunely adapted. In this environment, the graphic user interface to the network module is called Network Link Conditioner.

The results obtained by the network load experiment are shown in Figure 7.2.

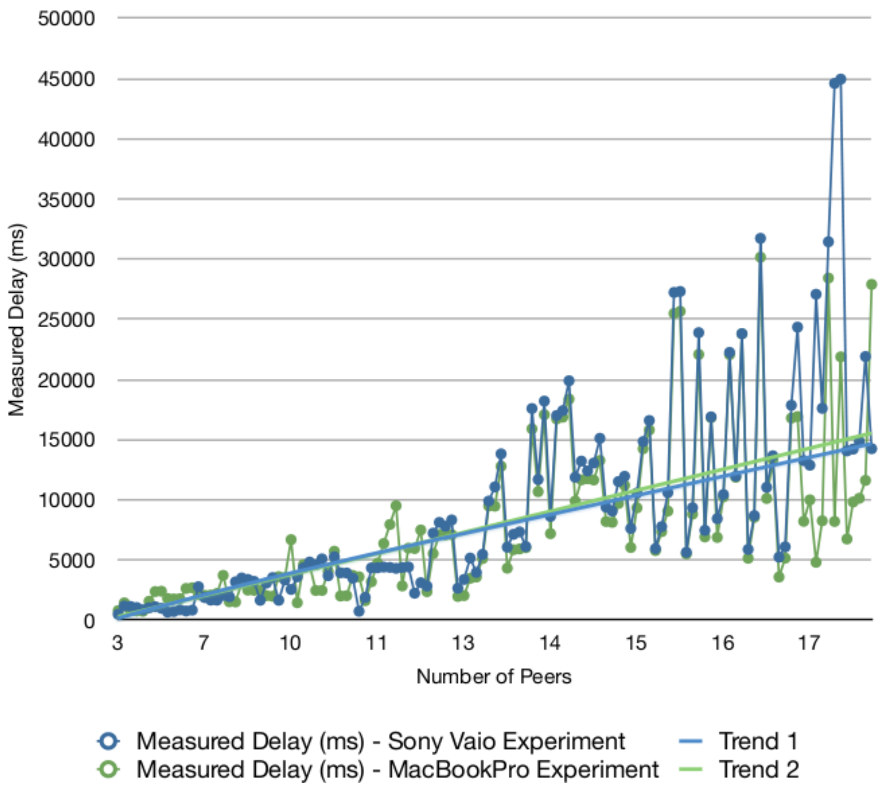


Figure 7.2: Network Load Performance Measurement

In order to decrease the number of messages per second sent by each peer, a linear interpolation related to avatar movements has been coded in the Implementor. Figure 7.3 shows the total number of message transmissions as the number of joining nodes grow, comparing the results obtained before and after the optimization. It is important to remark that the statistics shown in Figure 7.3 do not include the number of transmissions produced by the propagation mechanism of the underlying network layer.

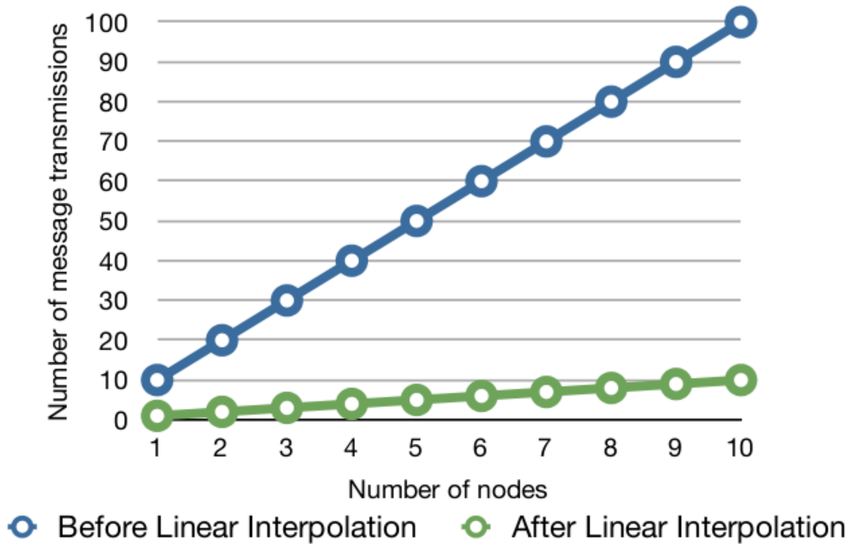


Figure 7.3: Total number of message transmissions for number of nodes

In terms of FPS (frames per second), the execution of the jME [31] Test Demo in a Fullscreen exclusive display mode reveals to be quite variable about performance: the MacBookPro runs the Demo ranging from 23 FPS to 71 FPS; on the iMac and the MacBook the FPS vary from 1 to 20 and from 3 to 9 respectively; the Sony Vaio executes the jME [31] Test Demo up to 11 FPS. When more than 4/5 users are connected to the same scene, the performance degradation in the execution of the Demo is clearly visible: only using the MacBookPro the difference was not so noticeable. In Table 7.2 are specified the performances in terms of FPS obtained varying the display resolution. The experiment has been performed running the jME [31] Test Demo in a shared LAN session with three participants.

During the execution of the tests, it has happened that the application crashed in a couple of occasions: in the first one, the event has occurred thirty minutes after the application has been started: the jME [31] Test Demo quit unexpectedly; in the second case, there was not a real crash, but a freeze related to the visualization

Table 7.2: Performance comparison

Hardware & OS	Display Mode	Min FPS	Max FPS
MacBookPro MAC OS X Lion	Fullscreen 1680x1050 32bpp	23	71
	Windowed 1024x768 24bpp	29	99
	Windowed 640x480 24bpp	38	142
iMac MAC OS X Lion	Fullscreen 1680x1050 32bpp	1	20
	Windowed 1024x768 24bpp	3	31
	Windowed 640x480 24bpp	8	71
MacBook MAC OS X Lion	Fullscreen 1280x800 32bpp	3	9
	Windowed 1024x768 24bpp	4	11
	Windowed 640x480 24bpp	8	22
Sony Vaio Linux Fedora 15	Fullscreen 1280x800 24bpp	2	11
	Windowed 1024x768 24bpp	5	16
	Windowed 640x480 24bpp	6	24

of the 3D Environment through the jMonkey Engine: it has not been possible to restore the proper functioning of the application.

Alongside the quantitative performances, also qualitative performances were deduced by involving two third party users. The first user can be considered a "medium/expert" user: 27 years old, he is not new to use a shell to run commands, is familiar with Windows and some Linux distributions, knows how to juggle between the different settings of an OS, has basic knowledge about Java programming. The second user can be targeted as a "newbie": 26 years old, he only knows the Windows Operating System, is unfamiliar with the command line, knows absolutely nothing in terms of programming. The two users were subjected to the same experiment: the *testShell.jar* and the *testJME.jar*, the JiveScript Shell Demo and the JiveScript jME [31] Demo, the User's Manual (6.2) and the Developer's Manual (6.1) have been provided them; they had to read carefully both the Manuals and try to perform the same operations described in the documents. In particular, they had to set up correctly all the necessary settings to run the Demos, start the JIVES applications by loading the scripts and execute some actions inside the Virtual Environment, such as moving around, managing the Inventory, combining items, chatting and trading with a NPC and a PC, saving and loading the application state.

The first experiment has been done by asking to execute JIVES and load the Shell Test Demo. Both users have no problem to start JIVES by launching the *java -jar testShell.jar* command in a terminal, although of course the time taken is definitely different: the "newbie" user took a few minutes to understand how to launch the terminal (the Command Prompt in his case), move into the right directory and start JIVES. Before loading the JiveScript Shell Demo, they were asked to configure the network settings to run the application as a simple Peer who wants to connect to a Rendezvous already running the same application. Both users have encountered difficulties in properly configuring the *networkConfigura-*

tion.xml, especially in understanding the meaning and usefulness of the various field of the XML file. The "newbie" user has also committed some mistakes in modifying the document according to the XML rules, since he has no knowledge about the syntax of the language.

Both users have loaded the script without any particular problem. Once loaded the script, they were asked to perform some operations inside the Virtual Environment. The actions that were performed correctly are: moving around the Virtual Environment, chatting with a NPC and a PC, obtaining items from the NPC, managing the Inventory, trading with a NPC, saving and loading the state application. When it came to combine two items, both users were in difficulty to understand the combination mechanism, but both have completed successfully the combination. Also trading with a PC was found to be unintuitive and difficult to properly execute. The "newbie" user in this case failed in executing correctly the trade.

Another factor that results from the experiment is the fact that the users had some difficulties in remembering the different commands needed to use the application.

The second experiment consisted in executing JIVES and load the jME [31] Test Demo. Thanks to the experience gained during the first experiment, both users have proven immediately to be much more responsive. Once started JIVES, they were asked to configure the network settings in order to start the Demo as a simple Peer. Both the "medium/expert" and the "newbie" users were able to configure the network settings by means the Nifty Graphical User Interface without any particular problem. When executing the jME [31] Demo, both users performed in the right way all the operations already tested in the first experiment. Unlike the Shell Demo, in this case the trading between PC has been considered much easier and intuitive; moreover they encountered no problem at all in performing the HotSpots combination thanks to the usability of the automatic alignment. Obviously if, due to Implementor design, the HotSpots were implicit, the time taken and the reasoning effort would have been higher. Both users were also asked to perform the combination without the combination helper: due to the lack of a reference system, they encountered some problems in managing the rotation and translation of the items in the Combination Screen. The time taken by the ”newbie” user to obtain a successful combination was a bit longer with respect to the other user. In both cases, however, the combination has been performed.

A third experiment has been performed, in which the users were asked to run the jME [31] Test Demo as a Java Applet. In this case, the experiment showed that the execution of JIVES using a browser is definitely the faster and easier one, especially from the point of view of the ”newbie” user. The only difficulty that has been encountered by the latter user was creating properly the *.java.policy* file in the right directory, in order to let the applet access the user’s machine. Note that users had free choice about which browser use: both executed the JIVES Applet by using Firefox.

The forth and last experiment was about the development of a JIVES application. Due to the short time available, the users were asked to write a new JiveScript, without the need of developing new Implementors, but by using the default ones provided by JIVES. The application had to be something very similar to the JiveScript jME [31] Demo, a Virtual Environment with few Non-Playing Characters and Artefacts. One day of time has been given to accomplish the request. Both users successfully installed the Netbeans [56] IDE and the plugin needed in order to benefit of the auto-completion feature provided by the JiveScript editor Netbeans plugin[15]. The ”medium/expert” user was able to develop a new JIVES application by writing a new JiveScript, using the default Implementors provided by the framework and the same meshes adopted in the development of the jME [31] Test Demo. The outcome is a 3D Virtual Environment which offers two different scenes, in which are located two Non-Playing Characters with a pre-defined dialogue and two Artefacts. The user has also defined a short plot, which consists in collecting some items, combining them together and trading them with the Non-Playing Characters in order to perform a specific action, possible only if the avatar owns particular items. At the contrary, the ”newbie” user, nevertheless he installed correctly all the necessary tools in order to build a new JIVES application, due to the lack of any knowledge in programming, failed to create a new JIVES application.

Both users, once they became familiar with the JIVES framework, were satisfied with the experience.

7.2 Evaluation

This section critiques the software solution that has been developed during the course of this project, evaluating the JIVES framework from a performance and functionality perspective. This will aim to determine whether the developed solution matches the requirements which were outlined in the Design section 5.1, and whether JIVES is performant enough to become a viable solution for the Deployment of Networked Java-based applications. A reflective analysis on the entire project as a whole will also be taken, discussing the validity of the approaches adopted, the reasoning behind these approaches and an appraisal of the fulfilment of the aims and objectives which were outlined in chapter 4.

Adopting the VESLOM [45] approach, it has been possible to build a framework that avoids a monolithic architecture. JIVES has been developed according to the VESLOM [45] Layers, obtaining in that way a good extensibility. Nevertheless JIVES is not totally extensible and modular, it allows the developer to define new Presentation and Network Implementors, while maintaining unchanged the Core. If the developer community will show interest in JIVES, the framework will greatly improve by means of the addition of new Implementors and custom actions in the Middleware Layer.

One of the key points specified in the design phase was the will to create a scripting language that would allow to develop easier JIVES applications. As seen in the test experiments and in our personal experience, we can assert that this goal was largely achieved: if the developer is not interested in creating new Implementors, he can develop a JIVES application making use of the JiveScript only. Once he becomes familiar with the scripting language, he will be able to write the application in an easier and faster way. Another feature of which JIVES can be proud is the possibility to define the whole logic of an application in the script, without having the need to write a single line of Java code.

Since the design phase, we set out to build a framework that could be distinguished from the others. This was possible thanks to the implementation of a fully integrated Item Inventory Management System, which permits to manage and combine the Inventory items in a way never seen before. Through the definition of several HotSpots for each item, JIVES offers the possibility to combine items in a more complete and meaningful way compared to other Virtual Environment systems. The test experiments have shown that combining the items together in a 3D Environment thanks to the HotSpot combination helper requires no practice at all in order to be performed with ease, the user has only to select by clicking them the proper HotSpots and complete the combination. The combination system has been implemented with a HotSpot combination helper in order to make it more user friendly: the goal has been largely achieved. Even when the users were forced to perform the combination without assistance of the combination helper, although with some difficulties the task has been completed. It is important to remark that the choice to use a combination helper is strictly related to the design of the Presentation Implementor.

The decision of a Peer-To-Peer architecture as the basis of the JIVES Networking Layer can be considered an innovative choice, due to the fact that the most of the existing Virtual Environment frameworks implement the network functionalities using a Client/Server technology. Thanks to the Peer-To-Peer technology integrated in the JIVES framework, a developer can create new Virtual Environment applications running on Internet at no server costs. Despite some difficulties in implementing the Network Implementor using the JXTA [54] Peer-To-Peer architecture, we can be satisfied due to the proper functioning of the JIVES network both when using a LAN connection and an Internet connection.

The choice of adopting the Java [51] language has proved to be the right one: the Java [51] compiler generates Java Virtual Machine code instead of machine code specific to the computer system the user is executing. The Java [51] compiled code is executable on any processor and system: it is sufficient to have the Java [51] Virtual Machine installed. Because of this JIVES reaches a great portability: it has been successfully executed on Windows, MAC OS X and Linux. Actually, it can also be considered a Web-VE when running as an applet.

From a technical point of view, the matter is a bit less enthusiastic. Prior to reducing the number of classes loaded by the *MultiInstanceEndpoint* JXSE [54] class-loader, unless the start-up time (the mean value of the measured times was 21.52) could be considered very good, the 3D visualization engine required big amounts of memory to run a 3D Virtual Environment application developed by using the JIVES framework. The optimization improved both the start-up time (the average time now is 15.56) and the memory consumption, reducing the latter by 50 %. Nevertheless, an high number of concurrent Peers can lead to system degradation. Such heaviness in the execution is not to be ascribed to JIVES but to the jME [31] visualization engine. Using a medium/high performance hardware, JIVES runs optimally along with jME [31]. In particular, minimum requirements are 1 GB RAM and a 512 MB graphics card; while recommended requirements are 2 GB RAM and a 1 GB graphics card.

Another problem regards the network scalability: when an high number of

Peers execute the same application a huge number of messages are sent through the network. In particular the main network performance problem resides in avatar movements and state updates that produce an high number of message exchanges. Even when an avatar doesn't move, for simplicity it sends an update message to the other users connected in the same scene. Nevertheless, experimental simulations showed up that the system scales up smoothly with ten users in a shared session. This result does not fit well within expectations: the goal was to be able to support at least 100 peers simultaneously. This is not to be regarded as a failure: by adopting the right network traffic reduction algorithms, such as dead reckoning [9], the network scalability can significantly improve. At the moment, due to the lag observed in the performed experiments, JIVES can not be considered capable of supporting the so-called "real-time" applications, such as First Person Shooters, with a lag that has to be below 150 ms. By adopting network optimizations techniques, it will be possible to reduce the network lag in order to extend the support also to this kind of applications. But this is the only limitation: from Inventory-based games to training simulations, such as simulations aimed to train the user in critical situations, to collaborative E-Learning Virtual Environments, such as interactive 3D class rooms, or professional working environments that can support collaborative projects, or virtual cities, museums, art galleries and labs, JIVES supports the creation of a large range of applications.

To perform a better estimation of the performance of the Peer-To-Peer solution, however, a small number of Peers is not sufficient. A cheap solution is performing the emulation of a wider network to understand the asymptotic behaviour of the network load. The trade-off consist in the fact that emulating a large number of applications on a single physical machine has the effect of a rapid degradation of the system responsiveness; this fact is going to drastically alter the trustiness of the evaluation as soon as the number of running Peers grows: it can be easily seen observing the increasing variance of the data plotted in Figure 7.2.

In any case, a linear trend clearly emerges from the regression line; this is an encouraging result if viewed in the perspective of the traditional Client/Server approach, where network load is exponential on the number of the connected Peers. An important observation to be done is that this is the worse case, in which n-1 Peers concur to contacting the n-th one: this tells us that *JXSEImplementor* is able to deliver the messages of 10 peers under 5 seconds on a network with the following characteristics: DSL, Downlink bandwidth 7 Mbps, Download Delay 100 ms, Uplink Bandwidth 2 Mbps, Uplink Delay 100 ms.

In terms of successful message delivery, JIVES has proved to be very efficient. Due to the choice of the JXTA [54] Architecture as basis for the Networking Layer, all the messages are sent adopting the TCP protocol, which guarantees delivery of messages without duplication or data loss. When it comes to jME [31] Test Demo, network traffic gained a great benefit from the introduction of an optimization such as the linear interpolation: the number of messages sent per second decreased by a 90 %.

From the point of view of executing JIVES as a Java Applet, the framework has pleasantly satisfied: there are no substantial differences between the performance as a Java Applet and the local counterpart. Moreover, the possibility to deploy a JIVES application as a Java Applet gives a fast and easy way to ensure that even less experienced users are able to run the application. Another important aspect is the fact that the applet works on most of the existing browsers, so that everyone can access it. In particular, as shown in Figure 7.4, the fact that the JIVES Applet can be used with Internet Explorer, Firefox, Chrome and Safari allows the framework to be properly executed by 92.8 % of web browser users.

The graph shows data updated to September 2011, extracted as median values from different sources, such as Net Applications [42], Statcounter [63], W3Counter [1], Wikimedia [73], Clicky [59].

The qualitative performances, obtained by means of the performed experi-

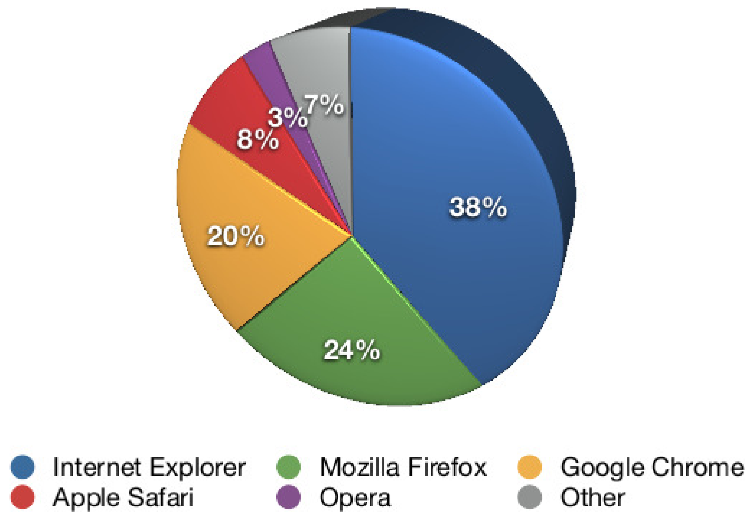


Figure 7.4: Web browser usage

ments, have shown a certain difficulty in using the Shell Test Demo, but this was expected and can be easily explained, since a textual Virtual Environment application is far less intuitive than the graphical counterpart. This has been demonstrated with the experiment that gave the possibility to the users to use the jME [31] Test Demo, of which they were fully satisfied. With respect to developing with JIVES, the experiments have proven that the framework is aimed at a "medium/expert" user, that has good knowledge in terms of Operating Systems and owns at least the basics of Java programming. This can be considered a success, due to the fact that the user does not need to be a real developer to create a JIVES application. However, JIVES may still be seen as attractive also by expert users, since they have the possibility to face new challenges when creating additional Implementors and custom actions that will extend the JIVES features.

In order to efficiently test the qualitative performances, the users who took part in usability experiments were asked to answer a short questionnaire, expressing their degree of satisfaction according to some qualitative indicators. The degree of satisfaction has been measured in a range from "unsatisfactory" to "very satisfactory". Each "X" represents a user's opinion. As shown in Table 7.3, the obtained results show that the framework allows to create applications, in particular 3D

Virtual Environments, that can be easily used by non-expert users. Moreover, the development of a new Virtual Environment application through JiveScript is feasible even for users who are not real developers.

Table 7.3: Qualitative performances assessment

Qualitative indicators	Deployment	Unsatisfactory	Satisfactory	Very satisfactory
Uptaking the User Manual	Shell Demo		X	X
	jME Demo			XX
	Applet			XX
Uptaking the Developer Manual	Shell Demo	X	X	
	jME Demo	X		X
	Applet		X	X
Ease in starting the application	Shell Demo		X	X
	jME Demo		X	X
	Applet			XX
Ease in configuring the network	Shell Demo	X	X	
	jME Demo			XX
	Applet			XX
User interaction during the execution of the application	Shell Demo		X	X
	jME Demo			XX
	Applet			XX
Ease in developing a new application	Shell Demo	X		X
	jME Demo	X		X
	Applet	X		X

Chapter 8

Conclusions and Future Work

This chapter provides a broad overview of the work which was undertaken throughout the project, and discusses the limitations which were encountered during the project process. The final section suggests the direction of future research and development which could be taken on the basis of this work.

8.1 Conclusions

The JIVES framework achieves almost all the goals set during the design phase: extensibility, ease of use and re-usability. Scalability can be considered achieved only partially, but in a satisfactory manner, due to the fact that there is a substantial room for improvement. From a technical point of view, JIVES can be significantly enhanced by integrating different optimization techniques. However, this aspect had been widely expected since JIVES was never meant to be specialized in the optimization of any particular requirement. Our aim was not to compare JIVES with other existing frameworks at a technical level, but was to provide something new in the Virtual Environment systems panorama: this aspect was largely achieved by creating a totally Open Source product, which offers a network architecture based on a Peer-To-Peer technology, its own scripting language and provides an important HotSpot-based Item Inventory Management System. JIVES can thus be considered without a doubt a success, and a significant starting point in the direction of creating a Virtual Environment framework that includes all the features and the optimizations needed in the development of such applications and that may be of interest to the developer community.

8.2 Future Work

Even if the project already looks promising and it is overall stable, there are many feature implementations and improvement possibilities; many of those would probably require a partial recoding, but the very basic structure of the software is not likely to change. The following list propose some key point that can make the project better, ordered from the one that is considered the most urgent, to the least.

- **Encryption of the local database**: as soon as an XTEA encryption adapter for DB4O [67] will be released, the mechanism of account authentication, secret key retrieval and snapshot encoding can be completed. This is the first step to allow the exploitation of JIVES in commercial application because, until now, the unencrypted save state is open to fraudulent modifications that will corrupt the distributed application state.

- **Cheat engine protection**: although not easy to exploit, the software is (as any resident application) exposed to direct and fraudulent RAM access in order to try to modify in-game variables values. Even if a correct synchronization with the persistence layer would greatly reduce this flaw, it might be reasonable to think about a possible server side checkpoint system to memorize and confront the client coherence states.

- **Network traffic reduction**: this is actually an Implementor issue, due to the fact that *JXSEImplementor* largely resorts to the use of broadcasting and due to the frequent update messages that the *JMEImplementor* needs in order to synchronize state and position of the peers. Actually the test implementors were not studied deeply for performance optimization as this is absolutely in scope of a commercial application, but not a must have for our framework test. A linear interpolation of the Playing Character's movements has already been implemented, allowing to reach a significant reduction in terms of network traffic rate: the number of messages per second sent by each Peer has decreased from 10 to 1. Even better results can be obtained by integrating in the JIVES framework some important message exchange reduction optimizations such as an Aura-based Interest Management [39] or a Predictive Interest Management [39] or a revisited version of the Delaunay triangulation [8].

- **Scalability improvement**: in order to extend the support of the JIVES framework also for massively-multiplayer applications and improve its scalability, some further optimizations can be explored, such as Dynamic Broadcast Tree technique [10].

- **Middleware structure**: at the moment the Implementors code is not subject to any guideline. There is simply the need of an engine and of a network interface; actually, there can be more rigid approaches, that will allow a polished Middleware structure (that needs to be defined) at the expense of a bit of the programmer's freedom. This should bring some advantages, like reducing the time of coding of new Implementors and keeping their code as much clear and integrated with the system as possible; thing, this one, that does not really apply to the test Implementors. With respect to this point, there will be the need to introduce an Audio Implementor, in order to have a text to speech feature for in-game dialogues and sounds in the 3D Environment. jME [31] can be extended to this, but this work implies to define a better Middleware structure.

- **User friendliness**: even if undoubtedly important, this feature was not a primary objective for the test Implementors. Although not impossible to understand and use, a command line interface is not what is generally considered friendly by a common user. A better, maybe graphical, interface for loading script, saving and restoring states and performing other common user operations would be preferable.

- **Concurrent activation action**: at present, there is no way in which two Peers can concur to the activation of an Activable entity; this is useful in the case that the distributed application state evolves differently according to who was able to perform the activation first; for instance, he or she might be the only user to receive a certain item from the entity. Usually the application evolves according to the logic of the script, which is the same on all the Peers but it's not shared among all the Peers. Ideally, a solution would be to include the information about the time of the entity activation in the *RemoteActiveNode* properties and handle the concurrency locally among all the playing character. This would require sharing a timestamp and keeping every Peer synchronized.

- **JiveScript generation tool**: the Netbeans plugin [15] to ease JiveScript editing is only a starting point. JiveScript entities definition, apart from the SIMs implementation, is very rigid. A possible line of development would consist in writing a Netbeans plugin that exposes a purely graphical interface which would allow the developer to choose which JIVES object to instantiate and parametrize, while the underlying code is procedurally generated.

- **Native resolution of JavaScript objects**: using the *javax.script* API allows to avoid taking care of security issues (like accessing protected Java scopes in Javascript) but, given that it does not expose JavaScript objects, this implies that native objects access has to be obtained using the Reflection

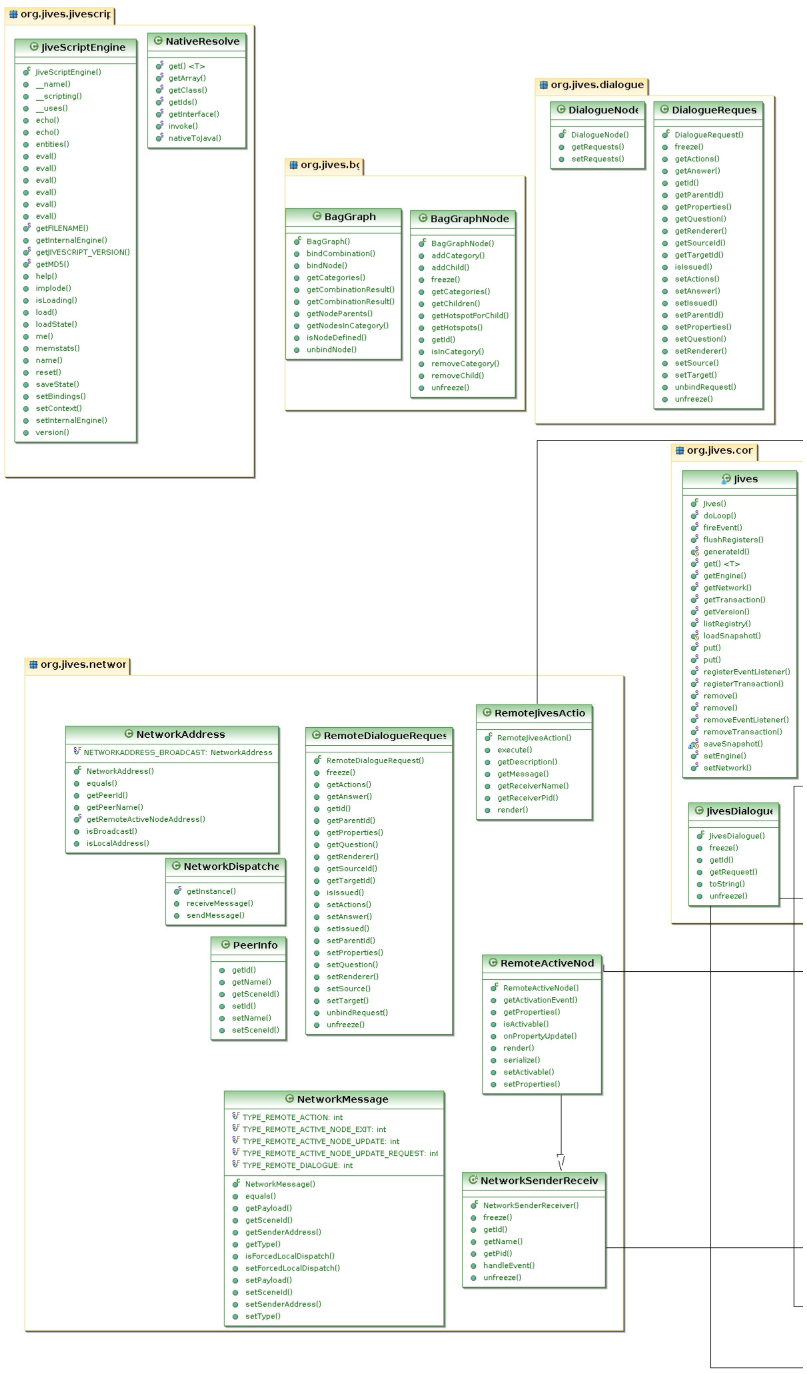
API. In the remote hypothesis that the low level of the scripting API will change, there is going to be some small recoding to be done.

- **Android support**: officially confirmed, the stable release of the jME3 [31] will support Android development. At the moment, the beta version of the SDK includes already the possibility to deploy an Android application. This feature can be exploited to extend the support of the JIVES framework also to Android, by creating a new Presentation Implementor that gives the possibility to run a JIVES application on a mobile device.

Appendix A

JIVES Class Diagram

Figure A.1 is the UML diagram of the most important classes inheritance showing public methods only. Core classes inherit the *JivesObjectIntf* interface, allowing them to exploit the JIVES registry. *Jives* class is static and does not need any instance. All actions extend *JivesAction* class, as *RemoteJivesAction* class does. The hierarchy of activables is defined inheriting the *Activable* abstract class, like *JivesActiveNode*, *JivesBag* and *JivesBagItem* classes do. Transactions are very generic because they are defined by an interface, the *TransactionIntf*, as also the Implementors are; respectively defined by *EngineImplementorIntf* and *NetworkImplementorIntf*. Events and Event Listeners are *JivesObjectIntf*, too. Implementing the *EventListenerIntf*, the *handleEvent()* method is served with every *JivesEvent* processed. Network receive events are handled this way by *NetworkSenderReceiver* class. Representations of remote entities implement this class to send and receive messages, like *RemoteActiveNode* does. Messages are specified by the *NetworkMessage* class. It has a sender address, realized by the *NetworkAddress* class, and a payload which can be any serializable entity, like an HashMap or maybe a dialogue containing one *RemoteDialogueRequest*. Other relevant packages are: *org.jives.jivescript* which includes the JiveScript interpreter; *org.jives.bg* that contains the Bag Graph; JIVES dialogue classes are restrained in *org.jives.dialogues*, while SIMs are contained in *org.jives.sim* and implemented by the respective JIVES Core class that delegates to them.


org.jives.jivescrip
JiveScriptEngine
JiveScriptEngine()
__name()
__scripting()
__uses()
echo()
echo()
entities()
eval()
eval()
eval()
eval()
eval()
eval()
getFILENAME()
getInternalEngine()
getJIVESCRIPT_VERSION()
getMD5()
help()
implode()
isLoading()
load()
loadState()
me()
memstats()
name()
reset()
saveState()
setBindings()
setContext()
setInternalEngine()
version()
NativeResolve
get() <T>
getArray()
getClass()
getIds()
getInterface()
invoke()
nativeToJava()
org.jives.bg
BagGraph
BagGraph()
bindCombination()
bindNode()
getCategories()
getCombinationResult()
getCombinationResult()
getNodeParents()
getNodesInCategory()
isNodeDefined()
unbindNode()
BagGraphNode
BagGraphNode()
addCategory()
addChild()
freeze()
getCategories()
getChildren()
getHotspotForChild()
getHotspots()
getId()
isInCategory()
removeCategory()
removeChild()
unfreeze()
org.jives.dialogue
DialogueNode
DialogueNode()
getRequests()
setRequests()
DialogueReques
DialogueRequest()
freeze()
getActions()
getAnswer()
getId()
getParentId()
getProperties()
getQuestion()
getRenderer()
getSourceId()
getTargetId()
isIssued()
setActions()
setAnswer()
setIssued()
setParentId()
setProperties()
setQuestion()
setRenderer()
setSource()
setTarget()
unbindRequest()
unfreeze()
org.jives.cor
Jives
Jives()
doLoop()
fireEvent()
flushRegisters()
generateId()
get() <T>
getEngine()
getNetwork()
getTransaction()
getVersion()
listRegistry()
loadSnapshot()
put()
put()
registerEventListener()
registerTransaction()
remove()
remove()
removeEventListener()
removeTransaction()
saveSnapshot()
setEngine()
setNetwork()
JivesDialogue
JivesDialogue()
freeze()
getId()
getRequest()
toString()
unfreeze()
org.jives.networ
NetworkAddress
NETWORKADDRESS_BROADCAST: NetworkAddress
NetworkAddress()
equals()
getPeerId()
getPeerName()
getRemoteActiveNodeAddress()
isBroadcast()
isLocalAddress()
NetworkDispatche
getInstance()
receiveMessage()
sendMessage()
PeerInfo
getId()
getName()
getSceneId()
setId()
setName()
setSceneId()
RemoteDialogueReques
RemoteDialogueRequest()
freeze()
getActions()
getAnswer()
getId()
getParentId()
getProperties()
getQuestion()
getRenderer()
getSourceId()
getTargetId()
isIssued()
setActions()
setAnswer()
setIssued()
setParentId()
setProperties()
setQuestion()
setRenderer()
setSource()
setTarget()
unbindRequest()
unfreeze()
RemoteJivesActio
RemoteJivesAction()
execute()
getDescription()
getMessage()
getReceiverName()
getReceiverPid()
render()
RemoteActiveNod
RemoteActiveNode()
getActivationEvent()
getProperties()
isActivable()
onPropertyUpdate()
render()
serialize()
setActivable()
setProperties()
NetworkMessage
TYPE_REMOTE_ACTION: int
TYPE_REMOTE_ACTIVE_NODE_EXIT: int
TYPE_REMOTE_ACTIVE_NODE_UPDATE: int
TYPE_REMOTE_ACTIVE_NODE_UPDATE_REQUEST: int
TYPE_REMOTE_DIALOGUE: int
NetworkMessage()
equals()
getPayload()
getSceneId()
getSenderAddress()
getType()
isForcedLocalDispatch()
setForcedLocalDispatch()
setPayload()
setSceneId()
setSenderAddress()
setType()
NetworkSenderReceiv
NetworkSenderReceiver()
freeze()
getId()
getName()
getPid()
handleEvent()
unfreeze()

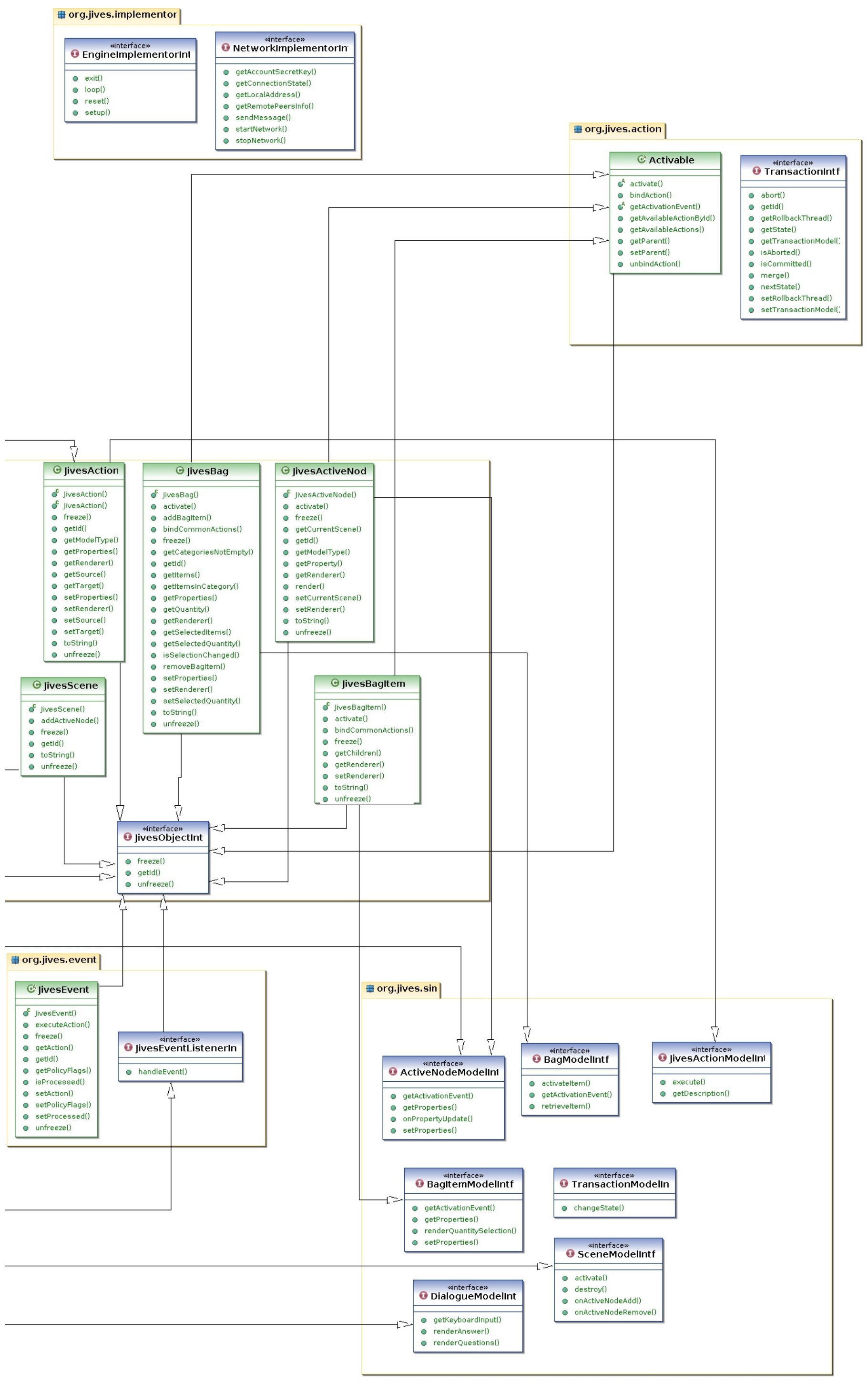


Figure A.1: JIVES Class Diagram

Appendix B

JiveScript Shell Demo

Code Snippet B.1: JiveScript Shell Demo

```
// JIVESCRIPT_VERSION = 0.2
reset();

__uses(org.jives.implementors.engine.shell);
__name("Jives Basic Demo");

echo("-------------------------------------------------------------------");
echo("  Welcome to Jives Basic Demo");
echo("-------------------------------------------------------------------");
echo("This is a textual adventure engine that uses Jives Shell implementor "
        + "and JiveScript.");
echo("Say \"help()\" for a list of available commands.");

// ----------------------------   Setup   -------------------------------- //
gui = org.jives.implementors.engine.shell.GUIState.getInstance();
dialogueModel = org.jives.implementors.engine.shell.SceneRenderState
        .getDialogueModel();
bagModel = new org.jives.implementors.engine.shell.BagRenderState("bag0");

// Define a listener
inventoryListener = new org.jives.events.JivesEventListenerIntf({
  handleEvent: function (jivesEvent) {
        action = jivesEvent.getAction();
        if (action instanceof org.jives.actions.PortalAction) {
            if (action.getCurrentSceneId().equals("scene0") && action
                .getNextSceneId().equals("scene1")) {
                org.jives.core.JivesScene.getMyself().getProperties()
                        .put("position", "Portal to Scene 0");
            }
            if (action.getCurrentSceneId().equals("scene1") && action
                .getNextSceneId().equals("scene0") ||
                action.getCurrentSceneId().equals("scene1") && action
                        .getNextSceneId().equals("scene2")) {
                org.jives.core.JivesScene.getMyself().getProperties()
                        .put("position", "Portal to Scene 1");
            }
            if (action.getCurrentSceneId().equals("scene2") && action
                .getNextSceneId().equals("scene1")) {
                org.jives.core.JivesScene.getMyself().getProperties()
                        .put("position", "Portal to Scene 2");
            }
        }
        if (action instanceof org.jives.actions.BagItemAction) {
            if(bag0.getQuantity("coin") < 2) {
                gui.notification("[Listener] Received " + action.getItem()
                        .getId() + ".");
                return false;
            } else {
                gui.notification("[Listener] You already have enough "
                + "coins.");
                return true;
            }
        }

        if (action.getId().equals("openCoffer")) {
            try {
                cofferOpened = Jives.get("cofferOpened", java.lang.String);
            } catch (e) {
                cofferOpened = "false";
            }
            if (cofferOpened == "false") {
                Jives.put("cofferOpened", "true");
                return false;
            } else {
                gui.notification("[Listener] The coffer is empty.");
                return true;
            }
        }

        return false;
    },
  getId: function () { return "inventoryListener"; }
});
Jives.registerEventListener(inventoryListener);

// Define bag0
bag0 = makeBag({
    bag0 : [
        {
            id: "apple",
            categories: [ "received" ],
            combines: [],
            model: new org.jives.implementors.engine.shell.BagItemRenderer(
                "An apple received trading with that strange guy."
            ),
            commonActionsRenderer: gui
        },
        {
            id: "coin",
            categories: [ "received" ],
            combines: [
                [ new org.jives.implementors.engine.shell
                        .HotspotRenderer("Useless hotspot"), null ]
            ],
            model: new org.jives.implementors.engine.shell.BagItemRenderer(
                "A coin received talking to that strange guy."
            ),
            commonActionsRenderer: gui
        },
        {
            id: "woodenStick",
            categories: [ "found" ],
            combines: [
                [ new org.jives.implementors.engine.shell
                        .HotspotRenderer("bottom"), null ],
                [ new org.jives.implementors.engine.shell
```

```
                        .HotspotRenderer("tip"), "stickWithWire" ]
            ],
            model: new org.jives.implementors.engine.shell.BagItemRenderer(
                "A strong and flexible wooden stick"
            ),
            commonActionsRenderer: gui
        },
        {
            id: "nylonWire",
            categories: [ "found" ],
            combines: [
                [ new org.jives.implementors.engine.shell
                        .HotspotRenderer("edge"), "stickWithWire" ],
                [ new org.jives.implementors.engine.shell
                        .HotspotRenderer("middle"), null ]
            ],
            model: new org.jives.implementors.engine.shell.BagItemRenderer(
                "A piece of nylon wire"
            ),
            commonActionsRenderer: gui
        },
        {
            id: "hook",
            categories: [ "found" ],
            combines: [
                [ new org.jives.implementors.engine.shell
                        .HotspotRenderer("eye"), "fishingRod" ],
                [ new org.jives.implementors.engine.shell
                        .HotspotRenderer("hook"), null ]
            ],
            model: new org.jives.implementors.engine.shell.BagItemRenderer(
                "A metal hook"
            ),
            commonActionsRenderer: gui
        },
        {
            id: "stickWithWire",
            categories: [ "found" ],
            combines: [
                [ new org.jives.implementors.engine.shell
                        .HotspotRenderer("wireEdge"), "fishingRod" ]
            ],
            model: new org.jives.implementors.engine.shell.BagItemRenderer(
                "A nylon wire tied to a wooden stick"
            ),
            commonActionsRenderer: gui
        },
        {
            id: "fishingRod",
            categories: [ "fishing" ],
            combines: [],
            model: new org.jives.implementors.engine.shell.BagItemRenderer(
                "A hand-crafted fishing rod"
            ),
            commonActionsRenderer: gui
        }
    ]
}, bagModel, bagModel);

// Define actions
bag0.bindCommonActions(gui);
dialogueItem = Jives.get("coin", org.jives.core.JivesBagItem);

// Define scene models
scene2Model = new org.jives.implementors.engine.shell.SceneRenderState(
    new org.jives.sim.SceneModelIntf({
        destroy: function () { },
        activate: function () {
            echo("\n\nSCENE 2:\n This room has a quiet pond with crystal "
                + "waters.");
        },
        onActiveNodeAdd: function (jivesActiveNode) {
            if (jivesActiveNode.getModelType().equals(org.jives.network
                .RemoteActiveNode)) {
                org.jives.implementors.engine.shell.SceneRenderState
                        .setupTradeAction(jivesActiveNode, bag0);
                org.jives.implementors.engine.shell.SceneRenderState
                        .setupRemoteDialogue(jivesActiveNode);
            }
        },
        onActiveNodeRemove: function (jivesActiveNode) { }
    })
);
scene2 = new org.jives.core.JivesScene("Scene 2", scene2Model);

scene1Model = new org.jives.implementors.engine.shell.SceneRenderState(
    new org.jives.sim.SceneModelIntf({
        destroy: function () { },
        activate: function () {
            echo("\n\nSCENE 1:\n This room has a door and there's a coffer"
                + "in the middle.");
        },
        onActiveNodeAdd: function (jivesActiveNode) {
            if (jivesActiveNode.getModelType().equals(org.jives.network
                .RemoteActiveNode)) {
                org.jives.implementors.engine.shell.SceneRenderState
                        .setupTradeAction(jivesActiveNode, bag0);
                org.jives.implementors.engine.shell.SceneRenderState
                        .setupRemoteDialogue(jivesActiveNode);
            }
        },
        onActiveNodeRemove: function (jivesActiveNode) { }
    })
);
scene1 = new org.jives.core.JivesScene("Scene 1", scene1Model);

scene0Model = new org.jives.implementors.engine.shell.SceneRenderState(
    new org.jives.sim.SceneModelIntf({
        destroy: function () { },
        activate: function () {
            echo("\n\nSCENE 0:\n This room has a door and there's a guy "
                + "in the southwest corner. "
                + "Use lookAround() to have a further object description.");
        },
        onActiveNodeAdd: function (jivesActiveNode) {
            if (jivesActiveNode.getModelType().equals(org.jives.network
                .RemoteActiveNode)) {
                org.jives.implementors.engine.shell.SceneRenderState
                        .setupTradeAction(jivesActiveNode, bag0);
                org.jives.implementors.engine.shell.SceneRenderState
                        .setupRemoteDialogue(jivesActiveNode);
            }
        },
        onActiveNodeRemove: function (jivesActiveNode) { }
    })
);
scene0 = new org.jives.core.JivesScene("Scene 0", scene0Model);

// ------------------------------- Scene 2 -------------------------//
// Door to scene 1 ---------------------------------------------------
scene2PortalTo1 = new org.jives.core.JivesActiveNode("scene2PortalTo1",
    new org.jives.implementors.engine.shell.ActiveNodeRenderer(
        org.jives.implementors.engine.shell.ActiveNodeRenderer.TYPE_PORTAL,
        "Portal to Scene 1"
    )
);
scene2.addActiveNode(scene2PortalTo1);
scene2PortalTo1.bindAction(
    new org.jives.actions.PortalAction(
        "scene2PortalTo1Action",
        scene2, scene1
    )
);

// Pond -----------------------------------------------------------
scene2Pond = new org.jives.core.JivesActiveNode("scene2Pond",
    new org.jives.implementors.engine.shell.ActiveNodeRenderer(
        org.jives.implementors.engine.shell.ActiveNodeRenderer.TYPE_NPC,
        "A pond"
    )
);
scene2.addActiveNode(scene2Pond);
scene2Pond.bindAction(
    new org.jives.actions.ActivateNodeWithItemAction("useFishingRodAction",
        bag0, scene2Pond,
        new org.jives.sim.JivesActionModelIntf({
            execute: function (jivesEvent) {
                action = jivesEvent.getAction();
                if (action.getBag().getSelectedItems().size() != 1) {
                  gui.notification("Select exactly one item from bag to "
                        + "proceed.");
                } else {
                  selectedItem = action.getBag().getSelectedItems().get(0)
                        .getId();
                  if (selectedItem.equals("fishingRod")) {
                    gui.notification("Fishing!");
                  } else {
                    gui.notification(selectedItem + " doesn't work here.");
                  }
                }
            },
            getDescription: function () { return "Fish"; },
            render: function (jivesRenderableIntf) { gui.notification(
                this.getDescription()); }
        })
    )
);

// ------------------------------- Scene 1 ----------------------//

// Door to scene 0 ------------------------------------------------
scene1PortalTo0 = new org.jives.core.JivesActiveNode("scene1PortalTo0",
    new org.jives.implementors.engine.shell.ActiveNodeRenderer(
        org.jives.implementors.engine.shell.ActiveNodeRenderer.TYPE_PORTAL,
        "Portal to Scene 0"
    )
);
scene1.addActiveNode(scene1PortalTo0);
scene1PortalTo0.bindAction(
    new org.jives.actions.PortalAction(
        "scene1PortalTo0Action",
        scene1, scene0
    )
);
```

```
// Door to scene 2 --------------------------------------------------
scene1PortalTo2 = new org.jives.core.JivesActiveNode("scene1PortalTo2",
    new org.jives.implementors.engine.shell.ActiveNodeRenderer(
        org.jives.implementors.engine.shell.ActiveNodeRenderer.TYPE_PORTAL,
        "Portal to Scene 2"
    )
);
scene1.addActiveNode(scene1PortalTo2);
scene1PortalTo2.bindAction(
    new org.jives.actions.PortalAction(
        "scene1PortalTo2Action",
        scene1, scene2
    )
);

// Coffer -----------------------------------------------------------
scene1Coffer = new org.jives.core.JivesActiveNode("scene1Coffer",
    new org.jives.implementors.engine.shell.ActiveNodeRenderer(
        org.jives.implementors.engine.shell.ActiveNodeRenderer.TYPE_NPC,
        "Coffer"
    )
);
scene1.addActiveNode(scene1Coffer);
scene1Coffer.bindAction(
    new org.jives.core.JivesAction("openCoffer",
        new org.jives.sim.JivesActionModelIntf({
            execute: function (jivesEvent) {
                bag0.addBagItem(Jives.get("woodenStick",
                        org.jives.core.JivesBagItem), 1);
                bag0.addBagItem(Jives.get("nylonWire",
                        org.jives.core.JivesBagItem), 1);
                bag0.addBagItem(Jives.get("hook",
                        org.jives.core.JivesBagItem), 2);

                gui.notification("Received a wooden stick, a nylon wire and "
                + "two hooks.");
            },
            getDescription: function () { return "Open coffer"; },
            render: function (jivesRenderableIntf) { gui.notification(
                this.getDescription()); }
        })
    )
);

// ------------------------------ Scene 0 ------------------------//

// Door to scene 1 ---------------------------------------------------
scene0PortalTo1 = new org.jives.core.JivesActiveNode("scene0PortalTo1",
    new org.jives.implementors.engine.shell.ActiveNodeRenderer(
        org.jives.implementors.engine.shell.ActiveNodeRenderer.TYPE_PORTAL,
        "Portal to Scene 1"
    )
);
scene0.addActiveNode(scene0PortalTo1);
scene0PortalTo1.bindAction(
    new org.jives.actions.PortalAction(
        "scene0PortalTo1Action",
        scene0, scene1
    )
);

// NPC ---------------------------------------------------------------
// Dialogue 0
scene0NPC0Dialogue0Root = makeDialogue({
    scene0NPC0Dialogue0Root : [
        {
            question: "Hello",
            answer: "Nice to meet you",
            actions: [
                new org.jives.actions.StartDialogueAction(
                    "scene0NPC0Dialogue0RootReq0Action",
                        "scene0NPC0Dialogue0Intro"
                )
            ]
        }
    ]
}, dialogueModel, gui);

makeDialogue({
    scene0NPC0Dialogue0Intro : [
        {
            question: "Where am I?",
            answer: "You are in Jives basic demo",
            actions: [
                new org.jives.actions.StartDialogueAction(
                   Jives.generateId(), "scene0NPC0Dialogue0Dialogues"
                )
            ]
        },
        {
            question: "Who are you?",
            answer: "I'm a non-playing character. I'm here to demonstrate"
                + "the dialogue system. Ask your questions.",
            actions: [
                new org.jives.actions.StartDialogueAction(
                    Jives.generateId(), "scene0NPC0Dialogue0Dialogues"
                )
            ]
        },
        {
            question: "Have a nice day",
            answer: "Goodbye"
        }
    ]
}, dialogueModel, gui);

makeDialogue({
    scene0NPC0Dialogue0Dialogues : [
        {
            question: "Can you give me an item to put in inventory?",
            answer: "I can; but for demonstration, the limit has been set "
                + "to two items only.",
            actions: [
                new org.jives.actions.BagItemAction(
                    Jives.generateId(), dialogueItem, bag0
                ),
                new org.jives.actions.StartDialogueAction(
                    Jives.generateId(), "scene0NPC0Dialogue0Dialogues"
                )
            ]
        },
        {
            question: "Can I trade items from my inventory?",
            answer: "Yes, we can trade. Activate me using the Trade action.",
            actions: [
                new org.jives.actions.StartDialogueAction(
                    Jives.generateId(), "scene0NPC0Dialogue0Dialogues"
                )
            ]
        },
        {
            question: "Can I trigger any action during a dialogue?",
            answer: "Of course; Here, an action is triggered and then "
                + "the dialogue continues",
            actions: [
                new org.jives.core.JivesAction("inDialogueAction",
                        new org.jives.sim.JivesActionModelIntf({
                    execute: function (jivesEvent) {
                        gui.notification("Light blinks and walls tremble "
                                + "for a moment...");
                    },
                    getDescription: function () {  return "In-dialogue "
                        + "action"; },
                    render: function (jivesRenderableIntf) {
                        gui.notification(this.getDescription()); }
                })),
                new org.jives.actions.StartDialogueAction(
                    Jives.generateId(), "scene0NPC0Dialogue0Dialogues"
                )
            ]
        },
        {
            question: "I've understood, thank you!",
            answer: "You're welcome!",
            actions: [
                new org.jives.actions.StartDialogueAction(
                    Jives.generateId(), "scene0NPC0Dialogue0Intro"
                )
            ]
        }
    ]
}, dialogueModel, gui);

// Bind dialogues to NPC
scene0NPC0 = new org.jives.core.JivesActiveNode("scene0NPC0",
    new org.jives.implementors.engine.shell.ActiveNodeRenderer(
        org.jives.implementors.engine.shell.ActiveNodeRenderer.TYPE_NPC,
        "Guy in the southwest corner"
    )
);
scene0.addActiveNode(scene0NPC0);
scene0NPC0.bindAction(
    new org.jives.actions.StartDialogueAction(
        "scene0NPC0Dialogue0RootAction",
        scene0NPC0Dialogue0Root.getId()
    )
);

// Bind trade to NPC
tradeItem = Jives.get("apple", org.jives.core.JivesBagItem);
catalog = new org.jives.implementors.engine.shell.ShopCatalog();
catalog.addEntry(tradeItem, -1, dialogueItem, 2);
org.jives.implementors.engine.shell.SceneRenderState.setupTradeAction(
scene0NPC0, bag0, catalog);

// ------------------------------- Start ----------------------------- //
// Trigger entrance action on scene 0
entranceAction = new org.jives.actions.PortalAction("entrancePortal", null,
        scene0);
entranceAction.execute(null);

// Prevent further scripting
__scripting(false);
```

Appendix C

JiveScript jME Demo

Code Snippet C.1: JiveScript jME Demo

```
// JIVESCRIPT_VERSION = 0.2
reset();

__uses(org.jives.implementors.engine.jme);
__name("Jives Basic Demo");

// ----------------------------   Setup   -------------------------------- //
gui = org.jives.implementors.engine.jme.GUIState.getInstance();
bagModel = new org.jives.implementors.engine.jme.BagRenderState("bag0");
bagCombiner = new org.jives.implementors.engine.jme.BagCombinerState();

// Set the model to use for the playing characters
org.jives.implementors.engine.jme.PlayingCharacterRenderer
        .setModelPath("Models/Oto/Oto.mesh.xml");

scene0Model = new org.jives.implementors.engine.jme.SceneRenderState(
    new org.jives.sim.SceneModelIntf({
        destroy: function () {
        },
        activate: function () {
        },
        onActiveNodeAdd: function (jivesActiveNode) {
            if (jivesActiveNode.getModelType().equals(org.jives.network
                .RemoteActiveNode)) {
                org.jives.implementors.engine.jme.SceneRenderState
                        .setupRemoteDialogue(jivesActiveNode);
                org.jives.implementors.engine.jme.SceneRenderState
                        .setupTradeAction(jivesActiveNode, bag0);
            }
        },
        onActiveNodeRemove: function (jivesActiveNodeArray) { }
    })
);
scene0 = new org.jives.core.JivesScene("Scene 0", scene0Model);
scene1Model = new org.jives.implementors.engine.jme.SceneRenderState(
    new org.jives.sim.SceneModelIntf({
        destroy: function () {
        },
        activate: function () {
        },
        onActiveNodeAdd: function (jivesActiveNode) {
            if (jivesActiveNode.getModelType().equals(org.jives.network
                .RemoteActiveNode)) {
                org.jives.implementors.engine.jme.SceneRenderState
                        .setupRemoteDialogue(jivesActiveNode);
                org.jives.implementors.engine.jme.SceneRenderState
                        .setupTradeAction(jivesActiveNode, bag0);
            }
        },
        onActiveNodeRemove: function (jivesActiveNodeArray) { }
    })
);
scene1 = new org.jives.core.JivesScene("Scene 1", scene1Model);
scene2Model = new org.jives.implementors.engine.jme.SceneRenderState(
    new org.jives.sim.SceneModelIntf({
        destroy: function () {
        },
        activate: function () {
        },
        onActiveNodeAdd: function (jivesActiveNode) {
            if (jivesActiveNode.getModelType().equals(org.jives.network
                .RemoteActiveNode)) {
                org.jives.implementors.engine.jme.SceneRenderState
                        .setupRemoteDialogue(jivesActiveNode);
                org.jives.implementors.engine.jme.SceneRenderState
                        .setupTradeAction(jivesActiveNode, bag0);
            }
        },
        onActiveNodeRemove: function (jivesActiveNodeArray) { }
    })
);
scene2 = new org.jives.core.JivesScene("Scene 2", scene2Model);

// Define a listener
inventoryListener = new org.jives.events.JivesEventListenerIntf({
  handleEvent: function (jivesEvent) {
        action = jivesEvent.getAction();
        if (action instanceof org.jives.actions.PortalAction) {
            if (action.getCurrentSceneId().equals("Scene 0")
                && action.getNextSceneId().equals("Scene 1")) {
                position = scene1PortalTo0.getProperties().get("position");
            }
            if (action.getCurrentSceneId().equals("Scene 1")
                && action.getNextSceneId().equals("Scene 0")) {
                position = scene0PortalTo1.getProperties().get("position");
            }
            if (action.getCurrentSceneId().equals("Scene 1")
                && action.getNextSceneId().equals("Scene 2")) {
                position = scene2PortalTo1.getProperties().get("position");
            }
            if (action.getCurrentSceneId().equals("Scene 2")
                && action.getNextSceneId().equals("Scene 1")) {
                position = scene1PortalTo2.getProperties().get("position");
            }
            org.jives.core.JivesScene.getMyself().getProperties()
                .put("position", position);
        }

        if (action instanceof org.jives.actions.BagItemAction) {
            if(bag0.getQuantity("coin") < 2) {
                gui.notification("[Listener] Received "
                        + action.getItem().getId() + ".");
                return false;
            } else {
                gui.notification("[Listener] You already have enough "
                        + "coins.");
                return true;
            }
        }

        if (action.getId().equals("openCoffer")) {
            try {
                cofferOpened = Jives.get("cofferOpened", java.lang.String);
            } catch (e) {
                cofferOpened = "false";
            }
            if (cofferOpened == "false") {
                Jives.put("cofferOpened", "true")
                return false;
            } else {
                gui.notification("[Listener] The coffer is empty.");
                return true;
            }
        }

        return false;
    },
  getId: function () { return "inventoryListener"; }
});
Jives.registerEventListener(inventoryListener);

// Define bag0
bag0 = makeBag({
    bag0 : [
    {
        id: "apple",
        categories: [ "received" ],
        combines: [],
        model: new org.jives.implementors.engine.jme.BagItemRenderer(
            "Models/apple/apple.mesh.xml", 0.1,
            "An apple received trading with that strange guy."
        ),
        commonActionsRenderer: gui
    },
    {
        id: "coin",
        categories: [ "received" ],
        combines: [
           [
               new org.jives.implementors.engine.jme.HotspotRenderer(
                  com.jme3.math.Vector3f(5, 5, 5), 2
               ), null
           ]
        ],
        model: new org.jives.implementors.engine.jme.BagItemRenderer(
            "Models/coin/coin.mesh.xml", 0.1,
            "A coin received talking to that strange guy."
        ),
        commonActionsRenderer: gui
    },
    {
        id: "woodenStick",
        categories: [ "found" ],
        combines: [
            [
               new org.jives.implementors.engine.jme.HotspotRenderer(
                  com.jme3.math.Vector3f(-3.42, -3.08, 0.19), 1
               ), null
            ],
            [
               new org.jives.implementors.engine.jme.HotspotRenderer(
                  com.jme3.math.Vector3f(9.5, 7.8, -0.03), 1
                ), "stickWithWire"
            ]
        ],
        model: new org.jives.implementors.engine.jme.BagItemRenderer(
            "Models/woodenStick/woodenStick.mesh.xml", 0.05,
            "A strong and flexible wooden stick"
        ),
        commonActionsRenderer: gui
    },
    {
        id: "nylonWire",
        categories: [ "found" ],
        combines: [
            [
                new org.jives.implementors.engine.jme.HotspotRenderer(
                    com.jme3.math.Vector3f(-10.29, 1, -0.91), 1
                ), "stickWithWire"
            ],
            [
                new org.jives.implementors.engine.jme.HotspotRenderer(
                    com.jme3.math.Vector3f(2.54, 1.25, 6.24), 2
                ), null
            ]
        ],
        model: new org.jives.implementors.engine.jme.BagItemRenderer(
            "Models/nylonWire/nylonWire.mesh.xml", 0.016,
            "A piece of nylon wire"
        ),
        commonActionsRenderer: gui
    },
    {
        id: "hook",
        categories: [ "found" ],
        combines: [
            [
                new org.jives.implementors.engine.jme.HotspotRenderer(
                    com.jme3.math.Vector3f(-1.8, -0.5, 1.6), 1
                ), "fishingRod"
            ],
            [
                new org.jives.implementors.engine.jme.HotspotRenderer(
                    com.jme3.math.Vector3f(-0.5, -0.5, 0), 0.5
                ), null
            ]
        ],
        model: new org.jives.implementors.engine.jme.BagItemRenderer(
            "Models/hook/hook.mesh.xml", 0.05,
            "A metal hook"
        ),
        commonActionsRenderer: gui
    },
    {
        id: "stickWithWire",
        categories: [ "found" ],
        combines: [
            [
                new org.jives.implementors.engine.jme.HotspotRenderer(
                    com.jme3.math.Vector3f(-0.42, -5.38, -2.59), 1
                ), "fishingRod"
            ]
        ],
        model: new org.jives.implementors.engine.jme.BagItemRenderer(
            "Models/stickWithWire/stickWithWire.mesh.xml", 0.05,
            "A nylon wire tied to a wooden stick"
        ),
        commonActionsRenderer: gui
    },
    {
        id: "fishingRod",
        categories: [ "fishing" ],
        combines: [],
        model: new org.jives.implementors.engine.jme.BagItemRenderer(
            "Models/fishingRod/fishingRod.mesh.xml", 0.05,
            "A hand-crafted fishing rod"
        ),
        commonActionsRenderer: gui
    }
 ]
}, bagModel, bagCombiner);

// Define actions
bag0.bindCommonActions(gui);
dialogueItem = Jives.get("coin", org.jives.core.JivesBagItem);

// ------------------------------- Scene 2 ------------------------------//

// Door to scene 1 ------------------------------------------------------
scene2PortalTo1 = new org.jives.core.JivesActiveNode("scene2PortalTo1",
    new org.jives.implementors.engine.jme.PortalRenderer(
        "Models/portal/portal.mesh.xml",
        new com.jme3.math.Vector3f(-60, 6, -70)
    )
);
scene2.addActiveNode(scene2PortalTo1);
scene2PortalTo1.bindAction(
    new org.jives.actions.PortalAction(
        "scene2PortalTo1Action",
        scene2, scene1
    )
);

// Pond ------------------------------------------------------------
scene2Pond = new org.jives.core.JivesActiveNode("scene2Pond",
    new org.jives.implementors.engine.jme.WaterNPCRenderer(
        "Models/pond/pond.mesh.xml",
        new com.jme3.math.Vector3f(-190, 7.4, 10),
        1.2
    )
);
scene2.addActiveNode(scene2Pond);
scene2Pond.bindAction(
    new org.jives.actions.ActivateNodeWithItemAction("useFishingRodAction",
        bag0, scene2Pond,
        new org.jives.sim.JivesActionModelIntf({
            execute: function (jivesEvent) {
                action = jivesEvent.getAction();
                if (action.getBag().getSelectedItems().size() != 1) {
                  gui.notification("Select exactly one item from bag "
                        + "to proceed.");
                } else {
                  selectedItem = action.getBag().getSelectedItems().get(0)
                        .getId();
                  if (selectedItem.equals("fishingRod")) {
                    gui.notification("Fishing!");
                  } else {
                    gui.notification(selectedItem + " doesn't work here.");
                  }
                }
            },
            getDescription: function () { return "Fish"; },
                render: function (jivesRenderableIntf) { gui.notification(
                    this.getDescription()); }
            })
        )
);

// ------------------------------- Scene 1 ------------------------------//

// Door to scene 0 ----------------------------------------------------
scene1PortalTo0 = new org.jives.core.JivesActiveNode("scene1PortalTo0",
        new org.jives.implementors.engine.jme.PortalRenderer(
            "Models/portal/portal.mesh.xml",
            new com.jme3.math.Vector3f(-170, 6, -90)
        )
);
scene1.addActiveNode(scene1PortalTo0);
scene1PortalTo0.bindAction(
        new org.jives.actions.PortalAction(
            "scene1PortalTo0Action",
            scene1, scene0
        )
);

// Door to scene 2 ----------------------------------------------------
scene1PortalTo2 = new org.jives.core.JivesActiveNode("scene1PortalTo2",
        new org.jives.implementors.engine.jme.PortalRenderer(
            "Models/portal/portal.mesh.xml",
            new com.jme3.math.Vector3f(-160, 6, -50)
        )
);
scene1.addActiveNode(scene1PortalTo2);
scene1PortalTo2.bindAction(
        new org.jives.actions.PortalAction(
            "scene1PortalTo2Action",
            scene1, scene2
        )
);

// Coffer --------------------------------------------------------------
scene1Coffer = new org.jives.core.JivesActiveNode("scene1Coffer",
        new org.jives.implementors.engine.jme.NPCRenderer(
            "Models/coffer/coffer.mesh.xml",
            new com.jme3.math.Vector3f(-180, 5, 10),
            0.2
        )
);
scene1.addActiveNode(scene1Coffer);
scene1Coffer.bindAction(
        new org.jives.core.JivesAction("openCoffer",
            new org.jives.sim.JivesActionModelIntf({
                execute: function (jivesEvent) {
                    bag0.addBagItem(Jives.get("woodenStick",
                            org.jives.core.JivesBagItem), 1);
                    bag0.addBagItem(Jives.get("nylonWire",
                            org.jives.core.JivesBagItem), 1);
                    bag0.addBagItem(Jives.get("hook",
                            org.jives.core.JivesBagItem), 2);

                    gui.notification("Received a wooden stick, a nylon wire and "
                            + "two hooks.");
                },
                getDescription: function () { return "Open coffer"; },
                render: function (jivesRenderableIntf) { gui.notification(
                    this.getDescription()); }
            })
```

```
    )
);

// ------------------------------ Scene 0 ------------------------------//

// Door to scene 1 ------------------------------------------------------
scene0PortalTo1 = new org.jives.core.JivesActiveNode("scene0PortalTo1",
    new org.jives.implementors.engine.jme.PortalRenderer(
        "Models/portal/portal.mesh.xml",
        new com.jme3.math.Vector3f(-100, 6, -10)
    )
);
scene0.addActiveNode(scene0PortalTo1);
scene0PortalTo1.bindAction(
    new org.jives.actions.PortalAction(
        "scene0PortalTo1Action",
        scene0, scene1
    )
);

// NPC ---------------------------------------------------------------
dialogueRenderer = new org.jives.implementors.engine.jme.nifty
        .DialogueController();
// Dialogue 0
scene0NPC0Dialogue0Root = makeDialogue({
    scene0NPC0Dialogue0Root : [
        {
            question: "Hello",
            answer: "Nice to meet you",
            actions: [
                new org.jives.actions.StartDialogueAction(
                    "scene0NPC0Dialogue0RootReq0Action",
                     "scene0NPC0Dialogue0Intro"
                )
            ]
        }
    ]
}, dialogueRenderer, dialogueRenderer);

makeDialogue({
    scene0NPC0Dialogue0Intro : [
        {
            question: "Where am I?",
            answer: "You are in Jives basic demo",
            actions: [
                new org.jives.actions.StartDialogueAction(
                   Jives.generateId(), "scene0NPC0Dialogue0Dialogues"
                )
            ]
        },
        {
            question: "Who are you?",
            answer: "I'm a non-playing character. I'm here to demonstrate "
                + "the dialogue system. Ask your questions.",
            actions: [
                new org.jives.actions.StartDialogueAction(
                    Jives.generateId(), "scene0NPC0Dialogue0Dialogues"
                )
            ]
        },
        {
            question: "Have a nice day",
            answer: "Goodbye"
        }
    ]
```

```
}, dialogueRenderer, dialogueRenderer);

makeDialogue({
    scene0NPC0Dialogue0Dialogues : [
        {
            question: "Can you give me an item to put in inventory?",
            answer: "I can; but for demonstration, the limit has been "
                + "set to two items only.",
            actions: [
                new org.jives.actions.BagItemAction(
                   Jives.generateId(), dialogueItem, bag0
                ),
                new org.jives.actions.StartDialogueAction(
                    Jives.generateId(), "scene0NPC0Dialogue0Dialogues"
                )
            ]
        },
        {
            question: "Can I trade items from my inventory?",
            answer: "Yes, we can trade. Activate me using the Trade action.",
            actions: [
                new org.jives.actions.StartDialogueAction(
                    Jives.generateId(), "scene0NPC0Dialogue0Dialogues"
                )
            ]
        },
        {
            question: "Can I trigger any action during a dialogue?",
            answer: "Of course; Here, an action is triggered and "
                + "then the dialogue continues",
            actions: [
                new org.jives.core.JivesAction("inDialogueAction",
                        new org.jives.sim.JivesActionModelIntf({
                                execute: function (jivesEvent) {
                                        gui.notification("Light blinks and "
                                                + "walls tremble "
                                                + "for a moment...");
                    },
                    getDescription: function () {  return "In-dialogue "
                        + "action"; },
                    render: function (jivesRenderableIntf) { gui
                        .notification(this.getDescription()); }
                })),
                new org.jives.actions.StartDialogueAction(
                    Jives.generateId(), "scene0NPC0Dialogue0Dialogues"
                )
            ]
        },
        {
            question: "I've understood, thank you!",
            answer: "You're welcome!",
            actions: [
                new org.jives.actions.StartDialogueAction(
                    Jives.generateId(), "scene0NPC0Dialogue0Intro"
                )
            ]
        }
    ]
}, dialogueRenderer, dialogueRenderer);

// Bind dialogues to NPC
scene0NPC0 = new org.jives.core.JivesActiveNode("scene0NPC0",
    new org.jives.implementors.engine.jme.NPCRenderer(
        "Models/Oto/Oto.mesh.xml",
        new com.jme3.math.Vector3f(-160, 11.8, 35),
            1
        )
);
scene0.addActiveNode(scene0NPC0);
scene0NPC0.bindAction(
    new org.jives.actions.StartDialogueAction(
            "scene0NPC0Dialogue0RootAction",
            scene0NPC0Dialogue0Root.getId()
    )
);

// Bind trade to NPC
tradeItem = Jives.get("apple", org.jives.core.JivesBagItem);
catalog = new org.jives.implementors.engine.jme.ShopCatalog();
catalog.addEntry(tradeItem, -1, dialogueItem, 2);
org.jives.implementors.engine.jme.SceneRenderState.setupTradeAction(
        scene0NPC0, bag0, catalog);

// ------------------------------ Start -------------------------------//
// Trigger entrance action on scene 0
entranceAction = new org.jives.actions.PortalAction("entrancePortal", null,
        scene0);
entranceAction.execute(null);

// Prevent further scripting
__scripting(false);
```

Appendix D

Server-side Rendezvous directory active page

Code Snippet D.1: Rendezvous directory active page

```php
<?php

define("APPLICATION_DIRECTORY","./apps/");

function validate_ipv6($value) {
  // has to contain ":" at least twice like in ::1 or 1234::abcd
  if (substr_count($value, ":") < 2) return false;
  // only 1 double colon allowed
  if (substr_count($value, "::") > 1) return false;
  $groups = explode(':', $value);
  $num_groups = count($groups);
  // 3-8 groups of 0-4 digits (1 group has to be at least 1 digit)
  if (($num_groups > 8) || ($num_groups < 3)) return false;
  $empty_groups = 0;
  foreach ($groups as $group) {
    $group = trim($group);
    if (!empty($group) && !(is_numeric($group) && ($group == 0))) {
      if (!preg_match('#([a-fA-F0-9]{0,4})#', $group)) return false;
    } else ++$empty_groups;
  }
  // the unspecified address :: is not valid in this case
  return ($empty_groups < $num_groups) ? true : false;
}

if(empty($_GET['op'])) {
  $_GET['op'] = "";
}
if($_GET['op'] == 'add' || $_GET['op'] == 'remove') {
  $match = false;
  if(!empty($_GET['addr'])) {
    preg_match('/[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}/',
        $_GET['addr'], $match);
  }
  if(!empty($_GET['addr2'])) {
    if(validate_ipv6($_GET['addr2'])) {
      // validating ipv6
      echo "IPv6 valid\n";
    } else {
      echo "IPv6 null\n";
    }

  }
  if(!empty($_GET['port'])) {
    preg_match('/[0-9]{1,5}/', $_GET['port'], $match);
  }
  if(!$match || empty($_GET['addr']) || empty($_GET['port'])
        || empty($_GET['program'])) {
    // not inserted also ipv6 (addr2) due to the fact that some peers
    //  may not have a IPv6 address
    $_GET['op'] = "";
  }
}
if($_GET['op'] == 'query') {
  if(empty($_GET['program'])) {
    $_GET['op'] = "";
  }
}

switch($_GET['op']) {
  case 'list' :
    $doc = new DOMDocument();
    $doc->load( 'rendezvous.xml' );

    $rendezvous_list = $doc->getElementsByTagName( "rendezvous" );
    foreach( $rendezvous_list as $rendezvous )
    {
      $program = $rendezvous->getAttributeNode('program')->value;
      $MD5 = $rendezvous->getAttributeNode('md5')->value;

      $IPv4S = $rendezvous->getElementsByTagName( "ipv4" );
      $IPv4 = $IPv4S->item(0)->nodeValue;

      $IPv6S = $rendezvous->getElementsByTagName( "ipv6" );
      $IPv6 = $IPv6S->item(0)->nodeValue;

      $ports = $rendezvous->getElementsByTagName( "port" );
      $port = $ports->item(0)->nodeValue;

      $PIDS = $rendezvous->getElementsByTagName( "pid" );
      $PID = $PIDS->item(0)->nodeValue;

      $times = $rendezvous->getElementsByTagName( "time" );
      $time = $times->item(0)->nodeValue;

      echo "$IPv4 - $IPv6 - $port - $PID - $program - $MD5 - $time\r\n";
      echo "<br />";
    }
    break;
  case 'add' :
    if (strcmp($_GET['token'],md5($_GET['pid'].$_GET['program']))==0) {

      $doc = new DOMDocument();
      $doc -> preserveWhiteSpace = false;
      $doc -> load( 'rendezvous.xml' );
      $doc -> formatOutput = true;
      $root = $doc->documentElement;
      $rendezvous_list = $root->getElementsByTagName('rendezvous');

      $rendezvous = $doc->createElement("rendezvous");
      $root->appendChild($rendezvous);

      $program = $doc->createAttribute("program");
      $rendezvous->appendChild($program);
      $programValue = $doc->createTextNode($_GET['program']);
      $program->appendChild($programValue);

      $MD5 = $doc->createAttribute("md5");
      $rendezvous->appendChild($MD5);

      $MD5Value = $doc->createTextNode($_GET['md5']);
      $MD5->appendChild($MD5Value);

      $IPv4 = $doc->createElement( "ipv4" );
      $text_IPv4 = $doc->createTextNode($_GET['addr']);
      $IPv4->appendChild($text_IPv4);
      $rendezvous->appendChild($IPv4);

      $IPv6 = $doc->createElement( "ipv6" );
      $text_IPv6 = $doc->createTextNode($_GET['addr2']);
      $IPv6->appendChild($text_IPv6);
      $rendezvous->appendChild($IPv6);

      $port = $doc->createElement( "port" );
      $text_port = $doc->createTextNode($_GET['port']);
      $port->appendChild($text_port);
      $rendezvous->appendChild($port);

      $PID = $doc->createElement( "pid" );
      $text_PID = $doc->createTextNode($_GET['pid']);
      $PID->appendChild($text_PID);
      $rendezvous->appendChild($PID);

      $time = $doc->createElement( "time" );
      $text_time = $doc->createTextNode($_GET['time']);
      $time->appendChild($text_time);
      $rendezvous->appendChild($time);

      $doc->save('rendezvous.xml');
      echo "Adding completed!";

    } else {

      echo "Adding failed. Token not corresponding.";

    }
    break;
  case 'remove' :
    if (strcmp($_GET['token'],md5($_GET['pid'].$_GET['program']))==0) {

      $doc = new DOMDocument();
      $doc -> preserveWhiteSpace = false;
      $doc -> load( 'rendezvous.xml' );
      $doc -> formatOutput = true;
      $root = $doc->documentElement;
      $rendezvous_list = $root->getElementsByTagName('rendezvous');
      $nodesToDelete=array();
      foreach ($rendezvous_list as $rendezvous) {
        $program = $rendezvous->getAttributeNode('program')->value;
        $MD5 = $rendezvous->getAttributeNode('md5')->value;
        $IPv4=$rendezvous->getElementsByTagName('ipv4')->item(0)->textContent;
        $IPv6=$rendezvous->getElementsByTagName('ipv6')->item(0)->textContent;
        $port=$rendezvous->getElementsByTagName('port')->item(0)->textContent;
        $PID=$rendezvous->getElementsByTagName('pid')->item(0)->textContent;
        $time=$rendezvous->getElementsByTagName('time')->item(0)->textContent;

        if($IPv4 == $_GET['addr'] && $IPv6 == $_GET['addr2'] && $port
                == $_GET['port'] && $PID == $_GET['pid'] && $program
                == $_GET['program'] && $MD5 == $_GET['md5']
                && $time != null) {
          $nodesToDelete[]=$rendezvous;
        }
      }
      foreach ($nodesToDelete as $node) $node->parentNode->removeChild($node);

      $doc->save('rendezvous.xml');
      echo "Removing completed!";

    } else {

      echo "Removing failed. Token not corresponding.";

    }
    break;
  case 'query' :
    $doc = new DOMDocument();
    $doc -> preserveWhiteSpace = false;
    $doc -> load( 'rendezvous.xml' );
    $doc -> formatOutput = true;
    $root = $doc->documentElement;
    $rendezvous_list = $root->getElementsByTagName('rendezvous');
    $nodesToDelete=array();
    foreach ($rendezvous_list as $rendezvous) {
      $program = $rendezvous->getAttributeNode('program')->value;
      $MD5 = $rendezvous->getAttributeNode('md5')->value;
      $IPv4=$rendezvous->getElementsByTagName('ipv4')->item(0)->textContent;
      $IPv6=$rendezvous->getElementsByTagName('ipv6')->item(0)->textContent;
      $port=$rendezvous->getElementsByTagName('port')->item(0)->textContent;
      $PID=$rendezvous->getElementsByTagName('pid')->item(0)->textContent;
      $time=$rendezvous->getElementsByTagName('time')->item(0)->textContent;

      if($program != $_GET['program'] || $MD5 != $_GET['md5']) {
        $nodesToRemove[]=$rendezvous;
      }
    }
    foreach ($nodesToRemove as $node) $node->parentNode->removeChild($node);

    $doc->save(APPLICATION_DIRECTORY.'rendezvous_'.$_GET['program'].'.xml');
    echo "Query completed!";
    break;
  case 'queryinactive' :
    $doc = new DOMDocument();
    $doc -> preserveWhiteSpace = false;
    $doc -> load( 'rendezvous.xml' );
    $doc -> formatOutput = true;
    $root = $doc->documentElement;
    $rendezvous_list = $root->getElementsByTagName('rendezvous');
    $nodesToDelete=array();
    $programFilesToKeep=array();
    $filesToDelete=array();
    foreach ($rendezvous_list as $rendezvous) {
      $program = $rendezvous->getAttributeNode('program')->value;
      $MD5 = $rendezvous->getAttributeNode('md5')->value;
      $IPv4=$rendezvous->getElementsByTagName('ipv4')->item(0)->textContent;
      $IPv6=$rendezvous->getElementsByTagName('ipv6')->item(0)->textContent;
      $port=$rendezvous->getElementsByTagName('port')->item(0)->textContent;
      $PID=$rendezvous->getElementsByTagName('pid')->item(0)->textContent;
      $time=$rendezvous->getElementsByTagName('time')->item(0)->textContent;

      if(($_GET['time'] - $time) > 60000) {
        $nodesToRemove[]=$rendezvous;
      }

      $programFilesToKeep[]=$program;
    }
    foreach ($nodesToRemove as $node) $node->parentNode->removeChild($node);

    $files = scandir(APPLICATION_DIRECTORY);

    foreach ($files as $index => $file) {
      foreach ($programFilesToKeep as $programFile) {
        if(strcmp($file,'rendezvous_'.$programFile.'.xml')==0) {
          unset($files[$index]);
        }
      }
    }

    foreach ($files as $file) {
      if(is_file(APPLICATION_DIRECTORY.$file)) {
        unlink(APPLICATION_DIRECTORY.$file);
      }
    }

    $doc->save('rendezvous.xml');
    echo "Query inactive rendezvous completed!";
    break;
  case 'updatetime' :
    if (strcmp($_GET['token'],md5($_GET['pid'].$_GET['program']))==0) {

      $doc = new DOMDocument();
      $doc -> preserveWhiteSpace = false;
      $doc -> load( 'rendezvous.xml' );
      $doc -> formatOutput = true;
      $root = $doc->documentElement;
      $rendezvous_list = $root->getElementsByTagName('rendezvous');
      foreach ($rendezvous_list as $rendezvous) {
        $program = $rendezvous->getAttributeNode('program')->value;
        $MD5 = $rendezvous->getAttributeNode('md5')->value;
        $IPv4=$rendezvous->getElementsByTagName('ipv4')->item(0)->textContent;
        $IPv6=$rendezvous->getElementsByTagName('ipv6')->item(0)->textContent;
        $port=$rendezvous->getElementsByTagName('port')->item(0)->textContent;
        $PID=$rendezvous->getElementsByTagName('pid')->item(0)->textContent;
        $time=$rendezvous->getElementsByTagName('time')->item(0)->textContent;

        if($IPv4 == $_GET['addr'] && $IPv6 == $_GET['addr2']
        && $port == $_GET['port'] && $PID == $_GET['pid']
        && $program == $_GET['program'] && $MD5 == $_GET['md5']
        && $time != null) {

          $rendezvous->removeChild($rendezvous->getElementsByTagName('time')
                ->item(0));

          $newtime = $doc->createElement( "time" );
          $text_newtime = $doc->createTextNode($_GET['time']);
          $newtime->appendChild($text_newtime);
          $rendezvous->appendChild($newtime);

          $response = "Update process completed! New time: ";
          $timetoprint = $_GET['time'];
          echo $response.$timetoprint;

          $doc->save('rendezvous.xml');

        }
      }

    } else {
      $response = "Update process not completed. Token not corresponding.";
      echo $response;

    }
    break;
  case 'gettoken' :
    $token = md5($_GET['pid'].$_GET['program']);
    echo $token;
    break;
  default :
    ?>

<h1>Rendezvous directory</h1>

<?php

}

  ?>
```

Bibliography

[1] Awio Web Services LLC. W3Counter - Free Realtime Web Analytics. http://www.w3counter.com/, Nov. 2011.

[2] Barbieri, T., and Paolini, P. Reconstructing Leonardo's ideal city - from handwritten codexes to webtalk-II: a 3D collaborative virtual environment system. In *Proceedings of the 2001 conference on Virtual reality, archeology, and cultural heritage* (New York, NY, USA, 2001), VAST '01, ACM, pp. 61–66.

[3] Belfore, II, L. A., Krishnan, P. V., and Baydogan, E. Common scene definition framework for constructing virtual worlds. In *Proceedings of the 37th conference on Winter simulation* (2005), WSC '05, Winter Simulation Conference, pp. 1985–1992.

[4] Biuk-Aghai, R. P., and Simoff, S. J. An integrative framework for knowledge extraction in collaborative virtual environments. In *Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work* (New York, NY, USA, 2001), GROUP '01, ACM, pp. 61–70.

[5] Blascovich, J. *Social influence within immersive virtual environments.* Springer-Verlag New York, Inc., New York, NY, USA, 2002, pp. 127–145.

[6] Blizzard Entertainment. World of Warcraft. http://eu.battle.net/wow/en/, Sept. 2011.

[7] Bosser, A. G. Massively multi-player games: matching game design with technical design. In *Proceedings of the 2004 ACM SIGCHI International Conference on Advances in computer entertainment technology* (New York, NY, USA, 2004), ACE '04, ACM, pp. 263–268.

[8] Buyukkaya, E., and Abdallah, M. Efficient triangulation for P2P networked virtual environments. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games* (New York, NY, USA, 2008), NetGames '08, ACM, pp. 34–39.

[9] Cai, W., Lee, F. B. S., and Chen, L. An auto-adaptive dead reckoning algorithm for distributed interactive simulation. In *Proceedings of the thirteenth workshop on Parallel and distributed simulation* (Washington, DC, USA, 1999), PADS '99, IEEE Computer Society, pp. 82–89.

[10] Chan, L., Yong, J., Bai, J., Leong, B., and Tan, R. Hydra: a massively-multiplayer peer-to-peer architecture for the game developer. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games* (New York, NY, USA, 2007), NetGames '07, ACM, pp. 37–42.

[11] CodeWeavers. WineHQ - Run Windows applications on Linux, BSD, Solaris and Mac OS X. http://www.winehq.org/, Sept. 2011.

[12] CPlusPlus.com. cplusplus.com - The C++ Resources Network. http://www.cplusplus.com/, Aug. 2011.

[13] Cube3. ActiveWorlds Managed .NET SDK. http://awmanaged.codeplex.com/, Oct. 2011.

[14] Dachselt, R., Hinz, M., and Meissner, K. Contigra: an xml-based architecture for component-oriented 3D applications. In *Proceedings of the sev-*

enth international conference on 3D Web technology (New York, NY, USA, 2002), Web3D '02, ACM, pp. 155–163.

[15] Dalpane, A. Netbeans plugin: Mozilla Rhino content assist. http://plugins.netbeans.org/plugin/39133/, Aug. 2011.

[16] Dalpane, A., and Segalini, S. Java Interactive Virtual Enviroment System - Official Web Site. http://sourceforge.net/projects/jives/, Sept. 2011.

[17] de Oliveira, J. C., Ahmed, D. T., and Shirmohammadi, S. Performance enhancement in mmogs using entity types. In *Proceedings of the 11th IEEE International Symposium on Distributed Simulation and Real-Time Applications* (Washington, DC, USA, 2007), DS-RT '07, IEEE Computer Society, pp. 25–30.

[18] Fabre, Y. A framework for mobile-agents embodied in X3D networked virtual environment. In *Proceedings of the eighth international conference on 3D Web technology* (New York, NY, USA, 2003), Web3D '03, ACM, pp. 113–122.

[19] Falko Braeutigam. Ozone - Java OODBMS. http://sourceforge.net/projects/ozone/, Oct. 2011.

[20] Fan, L., Taylor, H., and Trinder, P. Mediator: a design framework for P2P MMOGs. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games* (New York, NY, USA, 2007), NetGames '07, ACM, pp. 43–48.

[21] Free Software Foundation. Bash - GNU Project. http://www.gnu.org/software/bash/, Nov. 2011.

[22] Free Software Foundation. Welcome to GPLv3 - GPLv3. http://gplv3.fsf.org/, Sept. 2011.

[23] García, P., Montalà, O., Pairot, C., Rallo, R., and Skarmeta, A. G. MOVE: component groupware foundations for collaborative virtual environments. In *Proceedings of the 4th international conference on Collaborative virtual environments* (New York, NY, USA, 2002), CVE '02, ACM, pp. 55–62.

[24] Geeknet Inc. SourceForge.net: Find, Create, and Publish Open Source software for free. http://sourceforge.net/, Oct. 2011.

[25] Google Inc. Business Photos from Google. http://maps.google.com/help/maps/businessphotos/index.html, Sept. 2011.

[26] Google Inc. Google Maps with Street View. http://maps.google.it/intl/com/help/maps/streetview/, Sept. 2011.

[27] Halepovic, E., and Deters, R. The JXTA performance model and evaluation. *Future Gener. Comput. Syst. 21* (March 2005), 377–390.

[28] Hemminger, S. Netem. http://swik.net/netem, Nov. 2011.

[29] Hu, S.-Y., and Chen, J.-F. Von: a scalable peer-to-peer network for virtual environments. *Ieee Network 20*, 4 (2006), 22–31.

[30] Ingles, B. The future of Java™ game development. In *Proceedings of the 44th annual Southeast regional conference* (New York, NY, USA, 2006), ACM-SE 44, ACM, pp. 698–701.

[31] jMonkeyEngine. jMonkeyEngine 3.0 — Java OpenGL Game Engine. http://jmonkeyengine.com/, Aug. 2011.

[32] Khronos Group. OpenGL - The Industry Standard for High Performance Graphics. http://www.opengl.org/, Aug. 2011.

[33] Kienzle, J., Verbrugge, C., Bettina, K., Denault, A., and Hawker, M. Mammoth: a massively multiplayer game research framework. In *Proceedings of the 4th International Conference on Foundations of Digital Games* (New York, NY, USA, 2009), FDG '09, ACM, pp. 308–315.

[34] Kirner, T. G., Kirner, C., Kawamoto, A. L. S., Cantão, J., Pinto, A., and Wazlawick, R. S. Development of a collaborative virtual environment for educational applications. In *Proceedings of the sixth international conference on 3D Web technology* (New York, NY, USA, 2001), Web3D '01, ACM, pp. 61–68.

[35] Lee, D., Lim, M., Han, S., and Lee, K. ATLAS: A Scalable Network Framework for Distributed Virtual Environments. *Presence: Teleoper. Virtual Environ. 16* (April 2007), 125–156.

[36] Menchaca, R., Balladares, L., Quintero, R., and Carreto, C. Software engineering, HCI techniques and Java technologies joined to develop web-based 3D-collaborative virtual environments. In *Proceedings of the 2005 Latin American conference on Human-computer interaction* (New York, NY, USA, 2005), CLIHC '05, ACM, pp. 40–51.

[37] Michelson, B. M. Event-Driven Architecture Overview. (Originally Published February 2, 2006), Feb. 2011.

[38] Microsoft Corporation. Introduction to Direct3D 10 (SIGGRAPH 2007). http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=2858, Aug. 2011.

[39] Morgan, G., Lu, F., and Storey, K. Interest management middleware for networked games. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2005), I3D '05, ACM, pp. 57–64.

[40] Mozilla. Rhino - Javascript for Java. http://www.mozilla.org/rhino/, Aug. 2011.

[41] Mythic Entertainment. Ultima Online. http://www.uoherald.com/, Sept. 2011.

[42] NetApplications.com. Net Applications - Bringing Together Applications, Services and Partners. http://www.netapplications.com/, Nov. 2011.

[43] Nifty GUI. Nifty GUI - a Nifty GUI for your Java OpenGL/LWJGL application. http://nifty-gui.lessvoid.com/, Oct. 2011.

[44] Okanda, P., and Blair, G. OpenPING: a reflective middleware for the construction of adaptive networked game applications. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games* (New York, NY, USA, 2004), NetGames '04, ACM, pp. 111–115.

[45] Oliveira, M. Virtual environment system layered object model. In *Proceedings of the 2004 ACM SIGCHI International Conference on Advances in computer entertainment technology* (New York, NY, USA, 2004), ACE '04, ACM, pp. 194–202.

[46] Oliveira, M., Crowcroft, J., and Slater, M. An innovative design approach to build virtual environment systems. In *Proceedings of the workshop on Virtual environments 2003* (New York, NY, USA, 2003), EGVE '03, ACM, pp. 143–151.

[47] OpenWonderland Foundation. Open source 3D virtual collaboration toolkit: Open Wonderland. http://openwonderland.org/, Aug. 2011.

[48] Oracle. Java SE Desktop Technologies - Java 3D API. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138252.html, Aug. 2011.

[49] ORACLE. Java SE Documentation at a Glance. http://www.oracle.com/technetwork/java/javase/documentation/index.html, Oct. 2011.

[50] ORACLE. Java SE Downloads. http://www.oracle.com/technetwork/java/javase/downloads/index.html, Oct. 2011.

[51] ORACLE. java.com: Java + You. http://java.com/en/, Aug. 2011.

[52] ORACLE. javax.script - Java Platform SE 6. http://download.oracle.com/javase/6/docs/api/javax/script/package-summary.html, Oct. 2011.

[53] ORACLE. Jogl - Java.net. http://java.net/projects/jogl/, Aug. 2011.

[54] ORACLE. Jxta - Java.net. http://java.net/projects/jxta, Sept. 2011.

[55] ORACLE. lwjgl.org - Home of the Lightweight Java Game Library. http://lwjgl.org/, Aug. 2011.

[56] ORACLE. NetBeans IDE. http://www.netbeans.org/, Nov. 2011.

[57] PINHO, M. S., BOWMAN, D. A., AND FREITAS, C. M. Cooperative object manipulation in immersive virtual environments: framework and techniques. In *Proceedings of the ACM symposium on Virtual reality software and technology* (New York, NY, USA, 2002), VRST '02, ACM, pp. 171–178.

[58] QUAX, P., MONSIEURS, P., JEHAES, T., AND LAMOTTE, W. Using autonomous avatars to simulate a large-scale multi-user networked virtual environment. In *Proceedings of the 2004 ACM SIGGRAPH international conference on Virtual Reality continuum and its applications in industry* (New York, NY, USA, 2004), VRCAI '04, ACM, pp. 88–94.

[59] ROXR SOFTWARE LTD. Clicky - Web Analytics in Real Time. http://www.getclicky.com/, Nov. 2011.

[60] Russell, G., Donaldson, A. F., and Sheppard, P. Tackling online game development problems with a novel network scripting language. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games* (New York, NY, USA, 2008), NetGames '08, ACM, pp. 85–90.

[61] Second Life. Virtual Worlds, Avatars, free 3D chat, online meetings - Second Life Official Site. http://secondlife.com/, Aug. 2011.

[62] Sony Online Entertainment. EverQuest II Online Game - Official Game Site. http://everquest2.com/, Sept. 2011.

[63] StatCounter. StatCounter - Free Invisible Web Tracker, Hit Counter and Web Stats. http://www.statcounter.com/, Nov. 2011.

[64] The Eclipse Foundation. Eclipse - The Eclipse Foundation Open Source community website. http://www.eclipse.org/, Nov. 2011.

[65] Vani, V., and Mohan, S. Interactive 3D class room: a framework for Web3D using J3D and JMF. In *Proceedings of the 1st Amrita ACM-W Celebration on Women in Computing in India* (New York, NY, USA, 2010), A2CWiC '10, ACM, pp. 24:1–24:7.

[66] Verna, D., and Grumbach, A. Can We Define Virtual Reality? The MRIC Model. In *Virtual Worlds* (1998), J.-C. Heudin, Ed., vol. 1434 of *Lecture Notes in Computer Science*, Springer, pp. 29–41.

[67] Versant Corp. DB4O - Java & .NET Object Database - Open Source Object Database, Open Source Persistence, OODB. http://www.db4o.com/, Oct. 2011.

[68] Verstrynge, J. The JXTA Java™ Standard Edition Implementation Programmer's Guide, 2011.

[69] WEB3D CONSORTIUM. Basic, External Authoring Interface. http://www.web3d.org/x3d/content/examples/ExternalAuthoringInterface/index.html, Aug. 2011.

[70] WEB3D CONSORTIUM. Web3D Consortium - VRML Archives. http://www.web3d.org/x3d/vrml/, Aug. 2011.

[71] WEB3D CONSORTIUM. X3D for Developers. http://www.web3d.org/x3d/, Aug. 2011.

[72] WHITE, W., KOCH, C., GEHRKE, J., AND DEMERS, A. Better scripts, better games. *Queue 6* (November 2008), 18–25.

[73] WIKIMEDIA FOUNDATION. Wikimedia.org. http://www.wikimedia.org/, Nov. 2011.

[74] WORKFLOW, T., COALITION, M., AND NUMBER, D. Workflow Management Coalition White Paper - Events. *ReVision* (1999).

[75] ZWITSERLOOT, R., AND SPILKER, R. Project Lombok. http://projectlombok.org/index.html, Oct. 2011.