



PEPITO
IST-2001-33234
PEer-to-Peer Implementation and TheOry

Deliverable no: D3.8

**First progress report on
An interface between the Java VM and the
Distribution Subsystem**

REPORT VERSION: second

REPORT PREPARATION DATE: 2003.06.18, rev 2003.06.25

CLASSIFICATION: Public

DELIVERABLE NO: D3.8 DUE DATE: Month 18 DELIVERY DATE: Month 18

PROJECT START DATE: 2002.01.01 PROJECT DURATION: 36 months

RESPONSIBLE PARTNER: EPFL

PARTICIPATING PARTNERS: EPFL

PROJECT COORDINATOR: Swiss Institute of Technology Lausanne

PROJECT PARTNERS: EPFL Lausanne



**Project funded by the European Community under the
'Information Society Technologies' Programme (1998–
2002)**

Project Number: IST-2001-33234
Project Acronym: PEPITO
Title: PEer-to-Peer Implementation and TheOry
Deliverable No: D3.8
First progress report on
An interface between the Java VM and the Distribution
Subsystem
Due date: project month 18
Revised delivery date: 2003-03-25

Responsible Partner: EPFL
Participating Partners: EPFL

26th June 2003

Prepared by Martin Odersky and Stéphane Micheloud

Contents

1	Introduction	3
1.1	Objectives	3
1.2	Motivations and Vision	3
1.2.1	DSS	3
1.2.2	JDSS	3
1.3	Development environment	4
1.3.1	JNI	4
2	Current Achievements	6
2.1	System Architecture	6
2.2	JDSS Implementation	6
2.2.1	DSS.java	6
2.2.2	DSSMAP.java	7
2.2.3	DSSEnums.java	8
2.2.4	DSS.cc	8
2.2.5	DSSmaps.cc	9
2.3	Java example	9
2.4	Problems encountered	11
3	Planned work	12
4	Organizational changes	12
5	Relation to other workpackages in PEPITO	12

1 Introduction

This report presents the progress made in workpackage 3 (WP3) of the PEPITO project during the first year.

The report is organized as follows. In section 1.1 we state the objectives of workpackage 3. In section 1.2 we briefly present the motivations of the work done in this workpackage. Section 2 summarizes what has been achieved so far in this workpackage. In section 3 we present the tasks planned for the second year of activity. Section 4 summarizes the organizational changes which influence the work progress. In section 5, we highlight the connections between workpackage 3 and the other workpackages of the PEPITO project.

1.1 Objectives

The goal of workpackage 3 (WP3) is to integrate peer-to-peer (P2P) abilities into Oz and JAVA. The distributed middleware infrastructure is provided by the distribution system (DSS) developed in workpackage 4 (WP4).

This document focus on the implementation of an application programming interface (API) between the Java virtual machine (Java VM) and the DSS.

We expect the JAVA API to provide the same functionality as the C++ API defined in the DSS.

Investigations on the following topics are required to achieve the above objectives.

1. Distribution Subsystem (DSS).
2. Java Native Interface (JNI).

1.2 Motivations and Vision

In workpackage 3 we implement a JAVA API for the DSS called JDSS. For the realization of the JDSS we use the JAVA native interface (JNI) [6] technology from Sun Microsystems.

1.2.1 DSS

All languages/systems that fulfill the requirements defined for the DSS [1] - these are concurrency, reference security, data structuring, distinction between stateless and stateful data structures and automatic memory management - can be extended with transparent distribution by coupling the system to the DSS. For example programming languages such as JAVA, C#, Oz, Erlang, Caml can in principle use the DSS.

The DSS interface provides routines for globalization, memory management, buffering, marshaling, I/O and connectivity.

1.2.2 JDSS

At this time an OZ binding to the DSS exists. We would like to provide the capabilities of the DSS in a programming language which is in wide use. Due to its prominence in web services and distributed computing JAVA is a promising platform to make the services of the DSS widely available.

1.3 Development environment

Our development environment for this project is Red Hat Linux 8.0 and the following development tools are used to develop the JDSS software:

- Sun J2SE 1.4.1
- GNU C++ compiler 2.96
- GNU make 3.79.1

The JAVA platform is a programming environment consisting of the Java virtual machine (Java VM) and the Java Application Programming Interface (Java API). As a part of the JAVA VM implementation, the JNI [5] is a *two-way interface* that allows Java applications to invoke native code and vice versa. The developer can thus take advantage of the Java platform, but still utilize code written in other languages.

1.3.1 JNI

The main design goals of the JNI are *binary compatibility* among different Java VM implementations, *efficiency* to support time-critical code and *functional completeness* to enable native methods to accomplish useful tasks.

The JNI exposes the Java VM implementation details to the native code in four ways:

- By providing opaque references to JAVA objects, thereby hiding object implementation from native code.
- By placing a function table between the JAVA VM and native code and requiring all access of JAVA VM data to occur through these functions.
- By defining a set of native types to provide uniform mapping of JAVA types into platform-specific types.
- By providing flexibility to the JAVA VM vendor as to how object memory is handled in cases where the user expects continuous memory.

By using the JNI a Java application (or library) risks losing two benefits of the Java platform.

- Java applications that depend on the JNI can no longer readily run on multiple host environments. Indeed only the part of the application written in Java is portable, the rest must be recompiled for each new host environment.
- While the Java programming language is type-safe and secure, native languages such as C or C++ are not. As a result, extra care is required when writing applications using the JNI as a misbehaving native method can corrupt the entire application.

A general rule when using the JNI is to architect the applications so that native methods are defined in as few classes as possible. This entails a cleaner isolation between native code and the rest of the application.

The calling convention determines how a native function receives arguments and returns results. The JNI requires the native methods to be written in a specified standard calling convention on a given host environment. For example, the JNI follows the C calling convention on UNIX and the `stdcall` convention in the Win32 environment.

Errors made in JNI programming are different from exceptions that occur in the Java VM implementation. The JNI functions do not check for programming errors. Passing illegal arguments to JNI functions results in undefined behavior.

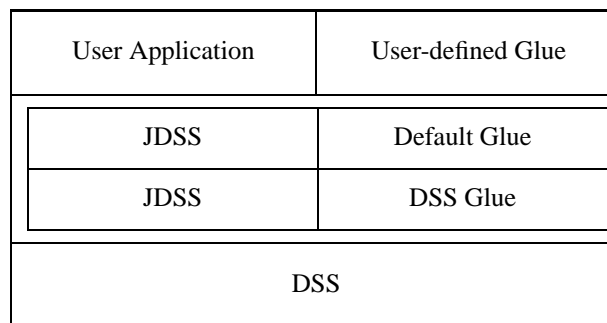
Finally the JNI does not rely on exception handling mechanisms in native programming languages as there is no standard mechanism commonly supported. Thus, JNI programmers are expected to check for and handle exceptions after each operation that can potentially throw an exception. It is extremely important to check, handle, and clear a pending exception before calling any subsequent JNI functions. Calling most JNI functions with a pending exception leads to undefined results.

2 Current Achievements

In this section we briefly present the system architecture of the JDSS and its implementation. Finally we outline some problems encountered so far in the development of the software.

2.1 System Architecture

The JDSS software is organized in layers. A layered architecture is designed as a hierarchy of client-server processes that minimizes interaction between layers. Each layer acts as a client for the module above it and acts as a server for the module below it.



As illustrated in the above diagram the software architecture is divided into three layers:

- The *application layer*: User applications written in JAVA and user-defined glue code belong to this layer.
- The *interface layer*: The JDSS itself is organized in two layers coded respectively in C++ and Java. Concretely the JDSS interface is composed of the JAVA library (`jdss.jar`) and the native library (`libJDSS.so`). Making both environment work together represents the main technical challenge of this workpackage.
- The *middleware layer*: The DSS (`libDSS.so`) provides transparent distribution to the host environment. This service layer hides the details and heterogeneity of the underlying platform.

2.2 JDSS Implementation

All JAVA sources of the JDSS are contained in the JAVA package `jdss`. In this report we present only the definitions of the JDSS needed for understanding the basic implementation mechanisms.

2.2.1 DSS.java

The JAVA class `DSS` constitutes the central part of the JDSS and provides the main functionality of the DSS API. While all JAVA methods in class `DSS` are declared `static`, we define exported and imported methods differently:

- Exported methods are declared as *native* methods and they use JNI to call the corresponding DSS methods in the DSS. The method `createEMU()` is given here as an example.
- Imported methods simply redirect the call to the actual adapter class (the class `MyDSSAdapter` is used here as default map). `disposeSiteRepresentative()` is given here as an example of an imported method.

Here is an outline of the definition of this class.

```

public final class DSS implements DSSEnums {

    private static DSSMAP map = new MyDSSAdapter() {};

    public static void addmap(DSSMAP map) {
        Debug.trace("{jdss.DSS}_addmap");
        DSS.map = map;
    }

    ////////////////////////////////// EXPORTS //////////////////////////////////

    /* Globalization, used when creating a distributed entity */

    public static native ProxyInterface createEMU(CM_Name cm,
        ProtocolName pn, Access_Architecture aa, RCalg GC_annot);

    // more ...

    ////////////////////////////////// IMPORTS //////////////////////////////////

    public static void disposeSiteRepresentative(DssReadBuffer buf,
        Site_Address addr) {
        map.disposeSiteRepresentative(buf, addr);
    }

    // more ...

    ////////////////////////////////// JNI specific stuffs //////////////////////////////////

    private static native void initIDs();

    static {
        System.loadLibrary("JDSS");
        initIDs();
    }

}

```

2.2.2 DSSMAP.java

The JAVA interface DSSMAP . java defines the functionality to be provided by the host environment for coupling the system to the DSS.

```

public interface DSSMAP extends DSSEnums {
    public Site_Address unmarshalSiteRepresentative(DssReadBuffer buf, Site_Name name);
    public void disposeSiteRepresentative(DssReadBuffer buf, Site_Address addr);
    public PstInContainerInterface createPstInContainer();
    public PstInContainerInterface loopBackOut2In(PstOutContainerInterface pstOut);
    public void GL_error(String msg);
}

```

```

    public void GL_warning(String msg) ;
    public void dssDebugInterface(DSS_AREA area, int param) ;
}

```

2.2.3 DSSEnums.java

The JAVA interface DSSEnums.java contains all enumeration objects corresponding to the C++ enumeration types declared in the include file dss_enum.hh of the DSS.

```

public interface DSSEnums {
    public AbsOp AO_NO_OP = new AbsOp() ;
    public AbsOp AO_CO_BIND = new AbsOp() ;
    public AbsOp AO_CO_BIND_VAR = new AbsOp() ;
    public AbsOp AO_CO_READ_VALUE = new AbsOp() ;
    public AbsOp AO_STATE_WRITE = new AbsOp() ;
    public AbsOp AO_STATE_READ = new AbsOp() ;
    public AbsOp AO_STATE_LOCK = new AbsOp() ;
    public AbsOp AO_STATE_UNLOCK = new AbsOp() ;
    public AbsOp AO_STATE_EXTRACT = new AbsOp() ;
    // many more . .
}

```

2.2.4 DSS.cc

The native method `initIDs` is called during the initialization of the class DSS and is responsible for initializing JNI specific datas and mappings.

```

JNIEXPORT jobject JNICALL Java_jdss_DSS_createEMU
(JNIEnv *env, jclass cls, jobject cm, jobject pn, jobject aa, jobject GC_annot) {
    DSS_TRACE("createEMU_called");
    // more ...
    ProxyInterface *val = createEMU((CM_Name) cm_val, (ProtocolName) pn_val,
        (Access_Architecture) aa_val, (RCalg) GC_annot_val);

    jclass clazz1 = env->FindClass(NAME_Proxy); // class Proxy implements ProxyInterface
    jmethodID mid = env->GetMethodID(clazz1, "<init>", "(JV)");
    DSS_ASSERT(mid != NULL);
    return env->NewObject(clazz1, mid, (jlong) val); // return result as Java Object
}

// more ...

JNIEXPORT void JNICALL Java_jdss_DSS_initIDs(JNIEnv *env, jclass cls) {
    DSS_TRACE("initIDs_called");

    const jclass clazz = env->FindClass(NAME_DSSEnums);
    DSS_ASSERT(clazz != NULL);
    initDecls(env, clazz);
    initMaps(env, clazz);
}

```

When the Java VM starts up, it constructs a list of directories that will be used to locate native libraries for application classes. The contents of the list are dependent upon the host environment and the VM implementation. For example, in the Win32 environment, the list of directories consists of the Windows system directories, the current working directory, and the entries in the PATH environment variable. Under Solaris or Linux the list of directories consists of the entries in the LD_LIBRARY_PATH environment variable.

2.2.5 DSSmaps.cc

The DSS middleware expects some functionality to be provided by the host environment. The binding of the imported JAVA methods with their corresponding DSS methods occurs in method `initMaps`.

```
void initMaps(JNIEnv *env, jclass cls) {
    DSSMAPS_TRACE("initMaps_enter");

    GL_error = GL_errorImpl;
    GL_warning = GL_warningImpl;

    unmarshalSiteRepresentative = unmarshalSiteRepresentativeImpl;
    disposeSiteRepresentative = disposeSiteRepresentativeImpl;

    DSSMAPS_TRACE("initMaps_leave");
}
```

For a better understanding of the mechanism we also give here the implementation of the method `disposeSiteRepresentativeImpl`.

```
void disposeSiteRepresentativeImpl(DssReadBuffer* buf, Site_Address* addr) {
    DSSMAPS_TRACE("disposeSiteRepresentative_called");

    JNIEnv *env = JNU_GetEnv();

    jclass clazz = env->FindClass(NAME_DSS);

    const char* sig = "(Ljcss/DssReadBuffer;Ljcss/Site_Address;)V";
    jmethodID mid = env->GetStaticMethodID(clazz, "disposeSiteRepresentative", sig);
    jobject obj = env->CallStaticObjectMethod(clazz, mid);
}
```

2.3 Java example

In order to illustrate the use of the JDSS we present here a simple JAVA application adapted from a test example provided in the DSS software distribution [9]. This example shows the expressiveness of the entity distribution model provided by the DSS which allows us to share entities between the two processes.

```
package examples.echo_server;

import java.net.InetAddress;
```

```

import jdss.*;
import utils.io_server.*;
import utils.map.*;

public class EchoServer {

    public static void main(String[] args) throws Exception {
        int iport = 9000;
        InetAddress ip = InetAddress.getByName("127.0.0.1");

        IOserver.initializeIO(ip, iport);
        OzVariable variable = new OzVariable();
        OzPort mp = new OzPort(variable);

        String str = new String(TicketTake.portToTicket(mp));
        System.out.println("Ticket:");
        System.out.println(str);

        for (;;) {
            while (variable.isFree()) {
                IOserver.establishConnections();
                IOserver.blocking_select();
            }
            OzData data = variable.getValue();
            if (OzData.isList(data)) {
                OzList lst = (OzList) data;
                OzData ele = lst.car();
                if (OzData.isPort(ele)) {
                    OzPort port = (OzPort) ele;
                    port.send(new OzAtom("Hello"));
                }
                if (OzData.isAtom(ele))
                    System.out.println("Received_" + ele.print());
                variable = (OzVariable) lst.cdr();
            }
            else
                DSS.GL_error("Var_is_not_a_list");
        } // for loop
    }
}

package examples.echo_server;

import java.net.InetAddress;

import jdss.*;
import utils.io_server.*;
import utils.map.*;

public class EchoClient {

```

```
public static void main(String[] args) throws Exception {
    if (args.length != 1) {
        DSS.GL_error("Wrong number of arguments");
    }

    String ticket = args[0];
    System.out.println("Connecting to:_" + ticket);
    int port = 9000;
    InetAddress ip = InetAddress.getByName("127.0.0.1");

    IOserver.initializeIO(ip, port);

    DSS.operateIntParam(DSS.DSS_STATIC, DSS.DSS_STATIC_LOG_PARAMETER,
        0, DSS.DLL_ALL.intValue());

    OzPort cp = TicketTake.ticketToPort(ticket.getBytes());
    OzVariable variable = new OzVariable();
    OzPort myPort = new OzPort(variable);

    cp.send(myPort);
    IOserver.establishConnections();

    while (variable.isFree()) {
        IOserver.blocking_select();
    }
    System.out.println("Done, received_" + variable.print());
}
}
```

2.4 Problems encountered

We summarize in this section the main problems encountered during the first project year:

- The adaptation of the JDSS to version 0.6 of the DSS required more effort than expected. We started a first implementation of the JDSS based on an early version of the DSS and then adapted it to version 0.6 which introduced many changes both in the API and in the implementation of the DSS.
- Debugging JAVA applications using JNI can be very cumbersome. JNI errors in native code generally result in JAVA core dump which are hard to work with. We thus decided to add a simple trace mechanism to the JDSS software.
- People developing the DSS and the JDSS are working in slightly different development environments. This led to some software configuration problems.

3 Planned work

The tasks planned for the second year of this workpackage are the following:

- More software testing. Only a subset of the functionality provided by the JDSS has been tested so far.
- Adaptation to the next DSS version(s). The current JDSS software is based on version 0.6 of the DSS.
- Automatic makefile generation.
- Generation of the API documentation using Javadoc. Javadoc [7] is the tool from Sun Microsystems for generating API documentation in HTML format from doc comments in source code.
- Project build on Windows using Cygwin [4]. Cygwin is a Linux-like environment for Windows. It consists of a DLL which acts as a Linux emulation layer providing substantial Linux API functionality and a collection of tools, which provide Linux look and feel.

4 Organizational changes

One team member left the project during the covered period: Dr. Christine Röckl, former workpackage manager for WP3, has left the EPFL in November 2002. Leila Rizkallah participated to the project as a summer student until end of June 2002.

5 Relation to other workpackages in PEPITO

In this section we show how this workpackage connects to the other workpackages of the project PEPITO.

- **WP4** This workpackage deals with the DSS and is strongly connected to workpackage 3. Indeed, the JDSS directly depends on the results of workpackage 4. Our contact persons for design and implementation questions about the DSS are Erik Klintskog and Zacharias El Banna. We initially planned to visit them in Sweden at the end of 2002, but the travel had to be delayed to some date in this year.
- **WP3** This workpackage deals with the experimentation of P2P programming abstractions using the Mozart and JAVA programming environments.
- **WP5** This workpackage is concerned with demonstrator applications which will use the results of workpackage 3.

References

- [1] Per Brand Erik Klintskog, Zacharias El Banna. *A Generic Middleware for Dynamic Intra Language Transparent Distribution*, 2003. <http://www.sics.se/pepito/>.
- [2] Free Software Foundation. *GNU make*, 1998.
<http://www.gnu.org/software/make/make.html>.
- [3] Rob Gordon. *Essential JNI*. Prentice Hall, 1998, ISBN 0-136-79895-0.
- [4] Red Hat. *Cygwin*, 2003.
<http://www.cygwin.com/>.
- [5] Sheng Liang. *The Java Native Interface*. Sun Microsystems, 1999, ISBN 0-201-32577-2.
<http://java.sun.com/docs/books/tutorial/native1.1/index.html>.
- [6] Sun Microsystems. *Java Native Interface*, 2002.
<http://java.sun.com/j2se/1.4/docs/guide/jni/>.
- [7] Sun Microsystems. *Javadoc Tool*, 2003.
<http://java.sun.com/j2se/javadoc/>.
- [8] Martin Odersky. *Report on the Programming Language Scala*, 2002.
<http://lampwww.epfl.ch/scala/>.
- [9] SICS. *The General Distributed SubSystem*, 2002.
<http://dss.sics.se/>.