



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

Deliverable no: D3.9

Final report on

An interface between the Java VM and the

Distribution Subsystem

REPORT VERSION: second

REPORT PREPARATION DATE: 2005.01.25

CLASSIFICATION: Public

DELIVERABLE NO: D3.9 DUE DATE: Month 39 DELIVERY DATE: Month 39

PROJECT START DATE: 2002.01.01 PROJECT DURATION: 39 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: D3.9
Final report on
An interface between the Java VM and the Distribution
Subsystem
Due date: project month 39
Revised delivery date: 2005-01-25

Responsible Partner: EPFL
Participating Partners: EPFL

11th March 2005

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	3
1.3.1	JNI	4
2	Current Achievements	5
2.1	System Architecture	5
2.2	JDSS Implementation	5
2.2.1	DSS_Object	5
2.2.2	Mediation_Object	7
2.2.3	DSSEnums	7
2.2.4	DSS.cc	7
2.3	Java example	8
2.4	Problems encountered	11
2.4.1	DSS	11
2.4.2	JNI	11
2.4.3	Development environment	11
3	Organizational changes	12
4	Relation to other workpackages in PEPITO	12

1 Introduction

This report presents the progress made during the last project year in workpackage 3 (WP3) of the PEPITO project. It can also be seen as the revisited (and final) version of deliverable D3.8 [?].

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 in this workpackage. Section 3 summarizes the organizational changes which have influenced the work progress. In section 4, 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 middleware.

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

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) [?] technology from Sun Microsystems.

1.2.1 DSS

All languages/systems that fulfill the requirements defined for the DSS [?] - 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 and is accessible to the programmer through a C++ API.

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

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 [?] 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.

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.

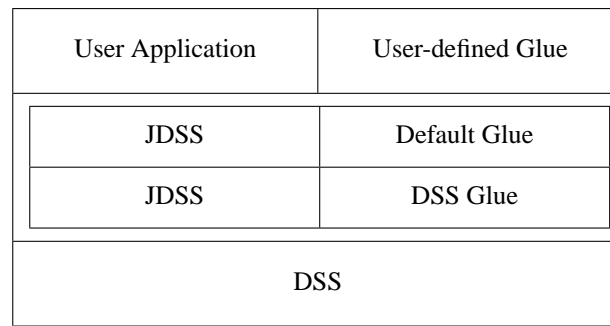
Finally the JNI does not rely on exception handling mechanisms in native programming languages as there is no standard mechanism commonly supported. Thus, JNI programmers are expected to check for and handle exceptions after each operation that can potentially throw an exception. It is extremely important 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 during the last project year in the development of the JDSS software.

2.1 System Architecture

The JDSS software is organized in three 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 layered into three parts:

- 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_Object

The JAVA class `DSS_Object` constitutes the central part of the JDSS and provides the main functionality of the DSS API. The native methods declared in class `DSS_Object` use JNI to call the corresponding C++ methods exported by the DSS (`dss_object.hh`).

Here is an outline of the definition of this class.

```
package jdss;
```

```

public final class DSS_Object {
    private static int refDssCpp;

    public DSS_Object(IoFactoryInterface io, ComServiceInterface sa, Mediation_Object mo) {
        this(io, sa, mo, false);
    }

    public DSS_Object(IoFactoryInterface io, ComServiceInterface sa,
        Mediation_Object mo, boolean sec) {
        refDssCpp = _DSS_Object(io, sa, mo, sec);
    }

    private native int _DSS_Object(IoFactoryInterface io, ComServiceInterface sa,
        Mediation_Object mo, boolean sec);

    public native boolean unmarshalProxy(
        AbstractEntity proxy, DssReadBuffer buf,
        ProxyUnmarshalFlag flag, AbstractEntityName cm);

    public native MutableAbstractEntity m_createMutableAbstractEntity(
        ProtocolName prot,
        Access_Architecture aa,
        /*int*/RCalg GC_annot);

    // more...

    public native DssThreadId m_createDssThreadId();

    // returns true if first can be said to be (globally) logically ordered before second.
    public native boolean m_orderEntities(AbstractEntity ae_first, AbstractEntity ae_second);

    // Periodically invoke to clean up internal DSS constructs
    public native void gcDssResources();

    // Exported by the DSS
    public native void heartBeat(int timePassedInMs);
    public native ParamRetVal operateIntParam(
        DSS_AREA area, DSS_AREA_ID id, int param, int arg);
    public native ParamRetVal operateStrParam(
        DSS_AREA area, DSS_AREA_ID id, int param, byte[] str);

    // ComInterface
    public native void connectionEstablished(DssSite ds, VirtualChannelInterface con);
    public native void virtualCircuitEstablished(DssSite ds, DssSite[] route);
    public native int installRTImonitor(DssSite ds, int lowLimit, int highLimit);
    public native void stateChange(DssSite ds, SiteState newState);
    public native void takeDownConnection(DssSite ds);
    public native ConnectivityStatus getChannelStatus(DssSite ds);
    public native void ext_sendMsg(DssSite ds, PstOutContainerInterface pst);
    public native void ext_marshallSiteRef(DssSite ds, DssWriteBuffer buf);
    public native CsSiteInterface ext_unmarshalSiteRef(DssReadBuffer buf);
    public native void kbr_start(DssSite ds, int k, int bits);

```

```

    public native KbrResult kbr_route(int key, PstOutContainerInterface value);
    public native void kbr_transferResp(PstOutContainerInterface resp);
    public native int kbr_getId();
}

```

2.2.2 Mediation_Object

The JAVA interface `Mediation_Object` defines the functionality to be provided by the host environment for coupling the system to the DSS (`dss_classes.hh`).

```

package jdss;

public interface Mediation_Object {
    public PstInContainerInterface createPstInContainer();

    public void GL_error(String format, Object[] args);
    public void GL_warning(String format, Object[] args);

    public void kbr_message(int key, PstInContainerInterface in);
    public void kbr_divideResp(int start, int stop, int n);
    public void kbr_newResp(int start, int stop, int n, PstInContainerInterface in);
    public void kbr_functional();
}

```

The implementation of this interface is provided by the class `utils.SimpleMedObj` which needs to be instantiated in the user application (see the Java example below).

2.2.3 DSSEnums

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

```

public interface DSSEnums {
    public AbsOp AO_NO_OP = new AbsOp();
    public AbsOp AO_OO_BIND = new AbsOp();
    public AbsOp AO_OO_BIND_VAR = new AbsOp();
    public AbsOp AO_OO_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 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.

2.3 Java example

In order to illustrate the use of the JDSS we present here a simple JAVA application adapted from a test example provided with the DSS software distribution [?]. 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.*;

// This example shows the expresivness of the entity distribution model. With
// simple means, we manage to share entities between the two processes.
//
// The two processes connects using the ticket mechanism described in the
// offer-take example. The server is now ready to receive two types of
// messages, atoms and ports. If a port is received, the server will echo
// hello on the port.

public class EchoServer {
    static {
        Debug.setDebug(true);
        Debug.trace("_");
        new DSS_Enums();
    }

    public static void main(String[] args) throws Exception {
        Debug.trace("_");
        Debug.trace("_");
        Debug.in("example.EchoServer:main");
        InetAddress ip = InetAddress.getByName("127.0.0.1");
        Debug.trace("IP=_"+ip);
    }
}

```

```

// Initializing the DSS
SimpleMedObj map = new SimpleMedObj(ip);
Debug.trace("dss_is_initialized, (SimpleMedObj)_map_" + map );

// Creates a port and a variable that
// The port will be exposed to remote processes
// using the "ticket" mechanism. Requests will come
// in on the stream, built by the port using the variable.
OzVariable variable = new OzVariable();
Debug.trace("variable_is_" + variable);
OzPort mp = new OzPort(variable);
Debug.trace("mp_" + mp);

// Creates marshaled representation of the port,
// and puts it on standard out.
String str = new String(TicketTake.entityToTicket(mp, map.getDSS()));
System.out.println("Ticket:");
System.out.println(str);

// A forever loop, waiting for data to arrive
// to the stream. The end of the stream is represented by
// the unbound variable v. As long as v is unbound, the
// process waits for incoming messages.
//
// When message arrives that binds the variable, the value
// is checked. If an Atom, the atom is printed. Else, if
// a port, an atom is sent on the port.
for (;;) {
    while (! variable.isDet()) {
        map.blocking_select();
    }
    OzData data = variable.getVal();
    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
        map.GL_error("Var_is_not_a_list", null);
}
}

package examples.echo_server;

```

```
import java.net.InetAddress;

import jdss.*;
import utils.*;

// This example shows the expressivness of the entity distribution model. With
// simple means, we manage to share entities between the two processes.
//
// The two processes connects using the ticket mechanism. Upon building the
// port, the taker then sends a new port on the built port. The new port will
// arrive at the receiving side, thus a fully functional port has been shared
// between the two processes.
//
// The server(i.e. the offerer of the contact port), sends the atom hello on
// the port. Upon receiving the atom from the server, the client terminates
// its process.

public class EchoClient {

    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
            System.err.println("Wrong number of arguments");
            System.exit(1);
        }
        String ticket = args[0];
        System.out.println("Connecting to:_" + ticket);
        InetAddress ip = InetAddress.getByName("127.0.0.1");

        // Initializing the DSS
        SimpleMedObj map = new SimpleMedObj(ip);

        int arg = DSS_Enums.DLL_ALL.intValue();
        DSS_Object dss = map.getDSS();
        dss.operateIntParam(DSS_Enums.DSS_STATIC,
            DSS_Enums.DSS_STATIC_LOG_PARAMETER, 0, arg);

        OzPort cp = (OzPort) TicketTake.ticketToEntity(ticket.getBytes(), dss);
        OzVariable variable = new OzVariable();
        OzPort myPort = new OzPort(variable);

        cp.send(myPort);

        while (! variable.isDet()) {
            map.blocking_select();
        }
        System.out.println("Done, _received_" + variable.print());
        System.exit(0);
    }
}
```

2.4 Problems encountered

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

- The adaptation of the JDSS from version 0.7 of the DSS to the current CVS version could not be finalized.
- 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.
- People developping the DSS and the JDSS are working in slightly different development environments. This led to some annoying software configuration problems.

2.4.1 DSS

At this stage the JAVA API doesn't yet provide the same functionality as the C++ API defined in the DSS. Version 0.7 of the DSS software has undergone many changes in its API to better exploit the object-oriented paradigm. This required a new adaptation of the JAVA API (see also deliverable D3.8 [?]).

2.4.2 JNI

Using with JNI implies a complex debugging process. We spent a lot of time to search for the origin of a problem and to solve it. Liang's book [?] was quite helpful in that time-consuming task.

2.4.3 Development environment

During the last project year we did several changes to our development environment:

- We first upgraded your operating system from Red Hat Linux 8.0 to Fedora Core 2 (and later to Fedora Core 3).
- We also updated our development tools, e.g. Sun J2SE 1.4.2 (1.4.1), GNU C++ compiler 3.4 (2.96), GNU make 3.80 (3.79.1).

GNU C++ 3.4 is much closer to full conformance to the ISO/ANSI C++ standard. This means, among other things, that a lot of invalid constructs which used to be accepted in previous versions are now rejected or produce warning messages. The DSS middleware was initially developed with GNU C++ 2.96.

3 Organizational changes

Vincent Porchet participated to the project as a summer student during the period August-December 2004. His contribution to the main achievements of the last project year was substantial.

4 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. Erik Klinskog was our main contact persons for design and implementation questions.
- **WP3** This workpackage deals with the experimentation of P2P programming abstractions using the Mozart and JAVA programming environments.
- **WP5** Finally this workpackage is concerned with demonstrator applications which will use the results of workpackage 3.

References

- [1] EPFL. *An interface between the Java VM and the Distribution Subsystem*, June 2003, Deliverable D3.8. <http://www.sics.se/pepito/deliverables.html>.
- [2] Per Brand Erik Klintskog, Zacharias El Banna. *A Generic Middleware for Dynamic Intra Language Transparent Distribution*, 2003. <http://www.sics.se/pepito/>.
- [3] Free Software Foundation. *GNU make*, 1998.
<http://www.gnu.org/software/make/make.html>.
- [4] Rob Gordon. *Essential JNI*. Prentice Hall, 1998, ISBN 0-136-79895-0.
- [5] Red Hat. *Cygwin*, 2003.
<http://www.cygwin.com/>.
- [6] 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>.
- [7] Sun Microsystems. *Java Native Interface*, 2002.
<http://java.sun.com/j2se/1.4/docs/guide/jni/>.
- [8] Sun Microsystems. *Javadoc Tool*, 2003.
<http://java.sun.com/j2se/javadoc/>.
- [9] Martin Odersky. *Report on the Programming Language Scala*, 2002.
<http://lampwww.epfl.ch/scala/>.
- [10] SICS. *The General Distributed SubSystem*, 2002.
<http://dss.sics.se/>.