



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

Deliverable no: D3.10

Progress report on Languages and Systems

REPORT VERSION: first

REPORT PREPARATION DATE: 2004.02.09

CLASSIFICATION: Public

DELIVERABLE NO: D3.10 DUE DATE: Month 24 DELIVERY DATE: Month 24

PROJECT START DATE: 2002.01.01 PROJECT DURATION: 36 months

RESPONSIBLE PARTNER: EPFL

PARTICIPATING PARTNERS: EPFL, INRIA, KTH, SICS, UCAM, UCL

PROJECT COORDINATOR: Swedish Institute of Computer Science, Kista

PROJECT PARTNERS: EPFL Lausanne, INRIA, KTH Stockholm, SICS Kista, UCAM Cambridge, UCL Louvain



**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: D3.10
Progress report on
Languages and Systems
Due date: project month 24
Delivery date: 2004-01-15

Responsible Partner: EPFL
Participating Partners: EPFL, INRIA, KTH, SICS, UCAM, UCL

9th February 2004

Prepared by Martin Odersky and Stéphane Micheloud, with input from Erik Klintskog, James Leifer, Christian Schulte and Peter Sewell.

Contents

1	Introduction	3
1.1	Objectives	3
1.2	Project Organization	3
2	Relations to other work packages	4
3	Mobile Computation	5
4	Resource Control	6
5	Mozart/Oz	8
6	Java/DACE/Scala	9
7	Dissemination	10
7.1	Publications	10
7.2	Web sites	10

1 Introduction

This report presents the progress made in work package 3 (WP3) of the PEPITO project during the first two years.

The report is organized as follows. In subsections 1.1 and 1.2 we state the objectives and organization of workpackage 3. In section 2, we highlight the connections between workpackage 3 and the other workpackages of the PEPITO project. The following sections summarize what has been achieved so far in this workpackage and what is planned for the last year of activity.

1.1 Objectives

This workpackage integrates peer-to-peer abilities into two languages, OZ and JAVA. The JAVA integration will be through a library and external interface. The OZ integration will be at the level of the MOZART virtual machine. Each language is designed to use the programming models of WP1 and WP2 in its own way and thus explore a different part of the programming space. Each language is implemented on top of its own virtual machine, which is responsible for all non-distributed execution. Each virtual machine will delegate all distributed execution to the distribution subsystem of WP4.

It is important to note that resource management is a language issue and not a distribution subsystem issue, and as such belongs in this workpackage. This can be shown as follows. Consider the *stationary object view* of a resource as an object that can be remotely accessed. The language or the application defines what messages can be passed to the object. The only role of the underlying distribution subsystem in this scenario (see WP4) is to allow secure message passing, e.g. with capabilities and encryption.

1.2 Project Organization

The tasks inside this workpackage are organized as follows:

Mobile computation (KTH)

- *Adding mobility to P2P.* This task is mainly to extend the OZ programming language with facilities for mobile computation (thread groups).
- Use of mobility in MOZART [S2].

Resource control (INRIA, UCAM)

- *Scope and type analysis.* Resources dependent on physical locations are dynamically linked to mobile code and computations. This presents challenges for scope and type analysis for the programming language which must ensure that mobile computations call resources in the correct way.
- Process calculi.

Mozart/Oz (SICS, UCL)

- *Adapt MOZART VM to distribution subsystem of WP4.* This task will integrate the Distribution Sybssystem developed in WP4 into the OZ programming language and its MOZART implementation.
- Extend OZ with programming abstractions of WP2.

Java/DACE/Scala (EPFL)

- *Java library classes*. This task will provide a frontend to the Distribution Subsystem developed in WP4 as a set of JAVA library classes.
- Adapt DACE [S3] and SCALA [S4] to use these classes.

2 Relations to other work packages

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

- **WP2** As a simplification, we currently assume that mobile computations move between completely known nodes. We expect results from workpackage 2 to make more realistic experiments, e.g. taking advantage of the distributed directory service in order to move mobile computations on an evolving network. Concretely most of the algorithms proposed in WP2 will be implemented by the P2PS library and tested with different applications. The obtained results can then be used as feedback to WP2.
- **WP4** This workpackage directly depends on the basic services provided by workpackage 4. Firstly, thread replication as one cornerstone of the implementation architecture for mobile computation uses the distribution support available in MOZART. Secondly, the development of the JAVA interface is tightly coupled to DSS implementation. Our contact persons for design and implementation questions about the DSS are Erik Klintskog and Zacharias El Banna.
- **WP5** This workpackage is concerned with demonstrator applications which will use the results of workpackage 3. Contacts exist with people working on a blog prototype; in particular Olov Stahl from the ICE laboratory at SICS is using JAVA JNI in his implementation work.

3 Mobile Computation

We have developed a model for strong mobility (execution state is mobile) with concomitant programming abstractions. The model is based on mobile thread and extends the distributed dataflow model underlying Oz and Mozart. A common implementation infrastructure based on thread replication and migration managers has been developed that provides the necessary support for the programming abstractions. A first feasibility study has been conducted by using a common test scenario for mobile applications.

This subworkpackage makes the general contribution of a model, programming abstractions, and sketch of an implementation architecture for thread-based strong mobility for a distributed dataflow language. In detail, the contributions are as follows:

- *Thread-based Mobility.* This deliverable contributes a model for strong mobility based on threads and dataflow languages. The model is an extension of a well-established model for distributed programming in a concurrent dataflow language. In particular the model insists on the fact that mobility is under the explicit control of the programmer.
- *Programming Abstractions and Application Scenarios.* Based on thread-based mobility, this deliverable contributes programming abstractions which capture common programming idioms in the construction of mobile applications. Prototype applications demonstrate how to use these abstractions for typical scenarios.
- *Sketch of Implementation Architecture.* All programming abstractions can be implemented by simple primitives for thread mobility. The deliverable identifies thread replication together with migration managers as basic building blocks for an architecture to implement thread-based mobility.
- *Initial Feasibility Test.* The programming abstractions have been implemented in a prototype on top of the MOZART programming system. The deliverable conducts a first feasibility test by running a standard test scenario for mobility [2].

A detailed description of the defined programming abstractions is available in deliverable D3.1 [1].

A first feasibility test of the model implemented in the Mozart programming system has been done to compare the performance of the mobile agent model built on top of thread-based mobility with the client-server model.

The current results need to be analyzed more intensively in some extreme application scenarios where P2P connections are facing unpredictable network configurations.

4 Resource Control

This deliverable is concerned with the theoretical aspects of the design of Acute, and more generally, with the problem of extending safe and expressive ML-like languages to meet the demands of distributed computing. Strong progress was made in this area during 2003.

This work enriches the Distributed Join calculus with dynamic channels whose messages are routed dynamically. The definition bound to a dynamic channel at a given location is the closest definition of this channel in the enclosing locations. A dynamic channel is available when there is a definition of this channel in the enclosing locations. A type system ensures statically that any reduction preserves the availability of dynamic channels. Thus even after several migrations, resources remain available.

A consequence of this work is the modeling of dynamic updates for channel definitions.

Part of this work could have been achieved in the pi-calculus. However this would have been more subtle since the association between senders and receivers is more dynamic than in the Join Calculus. In the pi-calculus, receivers may disappear. It is therefore more complex to distinguish between deadlock freedom and availability of receivers.

- ▷ The first aspect considered was the type- and abstraction-safety of marshalling, a fundamental requirement for distributed communication. Type abstraction is an important tool for modular development, preventing accidental dependencies on a module's implementation details. Values of an abstract type can only be operated on by its implementation, which will typically preserve some user invariants. Within a single program this is checked by conventional static typing, but what if values are marshalled and unmarshalled elsewhere? Simply ensuring that marshalling is type-safe is not enough: it should also be *abstraction-safe* by default, respecting abstraction boundaries (and hence preserving user invariants) across a distributed system.

We obtain a namespace for abstract types that is global, i.e. meaningful between programs, by hashing module declarations. We examine the scenarios in which values of abstract types are communicated from one program to another, and ensure, by constructing hashes appropriately, that the dynamic and static notions of type equality mirror each other. We use singleton kinds (Harper and Lillibridge) to express abstraction in the static semantics; abstraction is tracked in the dynamic semantics by colored brackets. These allow us to prove preservation, erasure, and coincidence results. Publications: [P6] and [P7].

- ▷ The second contribution concerned the control of dynamic rebinding of local resources when marshalling values that depend on their surrounding environment.

Static binding is desirable for most programming, but when we marshal a value it may have to be dynamically *rebound* to location-dependent resources, and also to local versions of modules – though sometimes we may want to ship the entire code-base it depends on. Thus we require expressive language constructs to specify what to ship and what to rebind. There may be version constraints that the value demands of its new context, and version declarations of the resources the receiver offers.

This argument leads us to consider the design of languages with marshalling of arbitrary language values to byte strings (above which various communication library modules can be expressed), type- and abstraction-safe unmarshalling, and a coherent notion of type equality between separately-developed programs that share some or no libraries, potentially of different versions.

Many existing languages do provide some form of *marshalling* or *serialization*, with a dynamic check at unmarshal-time. However, the constructs are often insufficiently expressive and not fully integrated with the rest of the language, leaving the programmer unable to control what is *shipped*, what is *checked* dynamically when unmarshalling, and how the shipped contents are *linked* in to the receiver. In particular, none deal satisfactorily with *type abstraction*, *rebinding* to local resources, and *version change*.

We discuss the design of programming languages for distributed computation, focusing on support for type-safe marshalling of arbitrary language values. In particular: (1) unmarshalling can involve *rebinding* to local resources; (2) values of *abstract types* can be communicated, and a globally-coherent notion of type equality ensures that unmarshalling respects abstraction; and (3) interoperation between separately-built programs with different *versions* of shared modules is supported, with fine-grain version control.

Submitted for publication: [P1], [P2], [P3] and [P8].

5 Mozart/Oz

DSS The Distribution Subsystem - DSS [P5] is a middleware implemented in WP4 of PEPITO. During the first year of the project, the basic functionality of the middleware has both been defined and implemented. This has resulted in a novel approach to distribution of programming language constructs that allows for simple, yet expressive instrumentation of distribution behaviour.

Oz-DSS Integration Work on the integration of Mozart and the DSS is in progress. In a prototype version the already existing distribution support for Mozart is replaced with the DSS, the system is referred to as Oz-DSS. Oz-DSS offers *transparent distribution* of all its language entities (something that was missing in Mozart). Furthermore, it implements correct handling of distributed operations, by preserving thread identities. Added to this is the possibility to define distribution strategy for single entities, catering for fine grained optimization of distributed applications.

The Oz-DSS platform has also been used to investigate the usage of P2P algorithms to enhance connectivity. Network problems that commonly halt a distributed computation, like moving processes and asymmetric network connections, can now be handled transparently. This work has been conducted jointly by Valentin Mesaros at UCL and Erik Klinskog together with Zacharias El-Banna at SICS.

The Oz-DSS platform has reached a mature state, as previous mentioned it offers notably more functionality than today's Mozart. However it still lacks some of the stability found in the well maintained Mozart system. Furthermore, Oz-DSS is based on a one year old version of Mozart, and thus lacks some of the later bug fixes and improvements. During autumn 2003 we have started the work on enhancing the latest available Mozart system with the distribution support of the DSS, thus making it a fully-fledged P2P programming platform. We expect to complete this work in 2004.

P2PS-DSS Relation Since P2PS offers P2P primitives, providing an implementation of different algorithms proposed in WP2, it can be used by the DSS to provide P2P services [P4].

The ongoing work that we conduct on the usage of P2P algorithms to improve connectivity in distributed systems is successful. As it was mentioned, this work was done in the Oz-DSS platform. Consequently, the P2PS library can, with little effort, be used by the OZ-DSS system. The benefits of the relation P2PS-DSS are twofold. First, new algorithms added to the P2PS library will immediately be available to the Oz-DSS system. Secondly, the P2PS module can make use of the advanced point-to-point communication abstractions provided by the DSS, thus avoiding redundant implementation efforts.

6 Java/DACE/Scala

With the implementation of an application programming interface (API) between the Java VM and the DSS, we provide the capabilities of the DSS in a programming language which is in wide use.

JDSS The Java platform is a programming environment consisting of the Java VM and the Java API. As a part of the Java VM implementation the JNI [3] is a *two-way interface* that allows Java applications to invoke native code and vice versa. Concretely we use JNI to implement the JDSS as a Java library which interacts with the DSS middleware itself written in C++.

The JDSS library is part of a system architecture consisting of three layers:

- The *application layer* contains user applications written in Java and user-defined glue code required by the underlying infrastructure.
- The *interface layer* - actually the JDSS library - is itself organized in two layers coded respectively in C++ and in Java.
- The *middleware layer* - actually the DSS - provides transparent distribution to the host environment.

A description of the implementation details is available in deliverable D3.8 [5].

We outline here some problems encountered so far in the development of a JAVA interface to the DSS:

- The adaptation of the JDSS to the different versions of the DSS required more effort than expected. Improvements in the development process are still possible, i.e. by intensifying the project coordination between the two development teams.
- 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. In fact problems faced here are related to the chosen technology. During the second year we could share some experiences on using JNI with people developing a demo application for WP5.
- People developing the DSS and the JDSS started working in slightly different development environments. This led to some software configuration problems. This situation has evolved positively during the second year so that we will soon be able to integrate the JDSS in the DSS software repository.

Further work efforts are needed on the following tasks:

- Adapting the JDSS to the current DSS version.
- Integrating the JDSS into the DSS software repository; this includes the automatic makefile generation.
- The functionality provided by the JDSS needs more extended testing.

We also plan to deliver:

- an API documentation to the JDSS using Javadoc. Javadoc [4] is the tool from Sun Microsystems for generating API documentation in HTML format from doc comments in source code.
- Examples illustrating the usage of the JDSS.

7 Dissemination

7.1 Publications

- [P1]Gavin Bierman, Michael Hicks, Peter Sewell, Gareth Stoye and Keith Wansbrough, *Dynamic Rebinding for Marshalling and Update, with Destruct-time λ* , ICFP 2003, Uppsala, August 2003, <http://www.cl.cam.ac.uk/users/pes20/dynbind2-short.ps>
- [P2]Gavin Bierman, Michael Hicks, Peter Sewell, Gareth Stoye and Keith Wansbrough, *Dynamic Rebinding for Marshalling and Update, with Destruct-time λ* , University of Cambridge, February 2004, UCAM-CL-TR-568, <http://www.cl.cam.ac.uk/users/pes20/UCAM-CL-TR-568.pdf>
- [P3]Gavin Bierman, Michael Hicks, Peter Sewell and Gareth Stoye, *Formalizing Dynamic Software Updating*, Proc. of USE 2003: the 2nd Intl. Workshop on Unanticipated Software Evolution, Warsaw, April 2003, <http://www.cl.cam.ac.uk/users/pes20/formalUpdate.pdf>
- [P4]Bruno Carton and Valentin Mesaros, *P2PS: A Peer-to-Peer Library Offering Primitives and Services for Applications, Software Library*, SICS, November 2003, MOGUL Archive, Mozart Programming System, <http://www.mozart-oz.org/mogul/>
- [P5]Erik Klinskog, Zacharias El Banna, Per Brand, *A Generic Middleware for Dynamic Intra Language Transparent Distribution*, SICS, 2003, <http://www.sics.se/pepito/>
- [P6]James J. Leifer and Gilles Peskine and Peter Sewell and Keith Wansbrough, *Global abstraction-safe marshalling with hash types*, Proc. 8th ICFP 2003, Uppsala, To appear, <http://pauillac.inria.fr/~leifer/research.html>
- [P7]James J. Leifer and Gilles Peskine and Peter Sewell and Keith Wansbrough, *Global abstraction-safe marshalling with hash types*, INRIA Rocquencourt, RR-4851, 2003, <http://pauillac.inria.fr/~leifer/research.html>
- [P8]James J. Leifer and Peter Sewell and Keith Wansbrough, *Marshalling: Abstraction, Rebinding, and Version Control*, INRIA Rocquencourt, 2003, To appear, <http://pauillac.inria.fr/~leifer/research.html>

7.2 Web sites

- [s1]SICS, *The General Distributed SubSystem*, SICS, 2002, <http://dss.sics.se/>
- [s2]Mozart Consortium, *The Mozart Programming System*, UdS, SICS, UCL, 2003, <http://www.mozart-oz.org/>
- [s3]Rachid Gerraoui and al., *The DACE Project*, EPFL, 2003, <http://www.d-a-c-e.com/>
- [s4]Martin Odersky and al., *The Scala Programming Language*, EPFL, 2003, <http://scala.epfl.ch/>

References

- [1] Christian Schulte Dragan Havelka, Seif Haridi. *Report on Mobile computation design*, 2003.
<http://www.sics.se/pepito/deliverables.html>.
- [2] D. Hagimont and L. Ismail. A performance evaluation of the mobile agent paradigm. In *Proc. 14th ACM SIGPLAN conference on OOPSLA*, pages 306–313. ACM Press, 1999.
<http://pauillac.inria.fr/~leifer/research.html>.
- [3] Sun Microsystems. *Java Native Interface*, 2002.
<http://java.sun.com/j2se/1.4/docs/guide/jni/>.
- [4] Sun Microsystems. *Javadoc Tool*, 2003.
<http://java.sun.com/j2se/javadoc/>.
- [5] Martin Odersky and Stéphane Micheloud. *First Progress Report on an Interface between the Java VM and the Distribution Subsystem*, 2003.
<http://www.sics.se/pepito/deliverables.html>.