



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

**Deliverable no: D3.11**  
**Final report on**  
**Languages and Systems**

REPORT VERSION: second    TOTAL NO OF PAGES: 12    CLASSIFICATION: Public

REPORT PREPARATION DATE: 2005.01.15, rev 2005.01.12

DELIVERABLE NO: D3.11    DUE DATE: Month 36    DELIVERY DATE: Month 36

ESTIMATED CUMULATIVE NO OF PERSON MONTHS REQUIRED: 1

PROJECT START DATE: 2002.01.01    PROJECT DURATION: 36 months

RESPONSIBLE PARTNER: EPFL

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

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: D3.11  
Final report on  
Languages and Systems  
Due date: project month 36  
Delivery date: 2005-01-15

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

25th January 2005

Prepared by Martin Odersky and Stéphane Micheloud, with input from James Leifer, Peter van Roy and Peter Sewell.

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Objectives . . . . .	3
1.2	Project Organization . . . . .	3
<b>2</b>	<b>Relations to other work packages</b>	<b>4</b>
<b>3</b>	<b>Mobile Computation</b>	<b>5</b>
3.1	P2PKit Services . . . . .	5
<b>4</b>	<b>Resource Control</b>	<b>6</b>
4.1	Acute Implementation . . . . .	6
4.2	Language Design . . . . .	6
<b>5</b>	<b>Mozart/Oz</b>	<b>8</b>
5.1	P2PS . . . . .	8
5.2	P2PKit . . . . .	8
5.3	Fault tolerance and the DSS . . . . .	8
<b>6</b>	<b>Java/DACE/Scala</b>	<b>9</b>
6.1	JDSS . . . . .	9
<b>7</b>	<b>Dissemination</b>	<b>10</b>
7.1	Publications . . . . .	10
7.2	Web sites . . . . .	11

## 1 Introduction

This report presents the progress made in work package 3 (WP3) of the PEPITO project during the third and last year.

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 in this work package during 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 [S3].

#### 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 [S4] and SCALA [S5] 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**
- **WP4**
- **WP5**

### 3 Mobile Computation

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.

The implementation of the model for strong thread-based mobile computations as described in the task Mobile computation [KTH:12] in workpackage 3 of the Technical Annex has been finished as planned for month 36.

The mobile computation design has been published at the Second International Oz/Mozart conference (MOZ 2004) in Charleroi, Belgium [P5].

For this deliverable we describe the work we have carried out to investigate useful concepts and language abstractions for programmers who wish to write robust, responsive, peer-to-peer languages. Our immediate goal is to make these abstractions easily accessible from the OZ language [S3].

#### 3.1 P2PKit Services

P2PKIT is a programmer's toolkit for Distributed Hash Table (DHT) based peer-to-peer middleware systems such as Chord [13], CAN [11], P2PS [10], Pastry [12], and Tapestry [8]. P2PKIT builds programmer-friendly abstractions on top of the primitive networking interfaces provided by these systems. An introduction and sample code for P2PKIT are available at <http://renoir.info.ucl.ac.be/twiki/bin/view/INGI/P2PKit>.

Our implementation is written in OZ and built on top of the P2PS peer-to-peer library. It is provided as a set of ozmake packages for easy installation.

P2PS is a DHT based peer-to-peer networking infrastructure. It uses the Tango algorithm, a variant of a distributed k-ary search tree (DKS), to efficiently route messages. Its administration overhead is small since the network self-optimises according to the messages sent by the application (i.e., it provides correction-on-use).

We are using P2PKIT to investigate and implement robust, decentralised services for replication, data storage and retrieval, distributed transactions, and so on. We are also writing applications, e.g., a graphical Control Centre, and deploying them on wide-scale distributed networks such as PlanetLab (<http://www.planet-lab.org/>) and Evergrow (<http://www.evergrow.org/>).

PlanetLab is a highly dynamic worldwide infrastructure that is sponsored by Intel. It consists of over 500 machines, hosted by over 230 sites, and spanning over 25 countries. Because its operation is reliant on the goodwill and responsiveness of local system and network administrators, it has frequent failures and network problems. Successfully running peer-to-peer applications on this infrastructure is a good test of robustness in a highly dynamic environment.

In Section 2 we describe the programmer's view of a peer in a peer-to-peer network and introduce P2PKIT services. In Section 3 we describe procedures for creating and joining a network of P2PKIT peers. In Section 4 we introduce P2PKit clients, and show how they can be used to simplify peer-to-peer applications. Section 5 describes how clients can install and upgrade services in a running P2PKit network. Services may also be loaded on-demand as described in Section 6. Section 7 describes P2PKIT notifiers. Section 8 describes how we use the P2PKIT abstractions to build a graphical Control Centre for peer-to-peer networks. And finally, we conclude with a discussion of further work.

## 4 Resource Control

This deliverable is concerned with the theoretical aspects of the design of ACUTE [S1], and more generally, with the problem of extending safe and expressive ML-like languages to meet the demands of distributed computing.

### 4.1 Acute Implementation

ACUTE [S1] is a test implementation of our current work on high-level programming languages for distributed computation. For motivation and a description of the language, see [P10].

The main priority for the implementation is to be close to the semantics, to make it easy to change as the definition changed, and easy to have reasonable confidence that the two agree, while being efficient enough to run moderate examples. An automated testing framework helps ensure the two are in sync.

The runtime is essentially an interpreter over the abstract syntax, finding redexes and performing reduction steps as in the semantics. For efficiency it uses closures and represents terms as pairs of an explicit evaluation context and the enclosed term to avoid having to re-traverse the whole term when finding redexes. Marshalled values are represented simply by a pretty-print of their abstract syntax. Numeric hashes use a hash function applied to a pretty-print of their body; it is thus important for this pretty-print to be canonical, choosing bound identifiers appropriately. ACUTE threads are reduced in turn, round-robin. A pool of OS threads is maintained for making blocking system calls. A genlib tool makes it easy to import (restricted versions of) OCaml libraries, taking OCaml .mli interface files and generating embeddings and projections between the OCaml and internal ACUTE representations.

On top of ACUTE, we have written libraries for TCP connection management and string messaging, local and distributed channels, remote function invocation, as well as a bigger libraries that implements a process language, namely Mobile Ambients. Our experience suggests that our lightweight extension to ML suffices to enable sophisticated distributed infrastructure to be programmed as simple libraries.

Our prototype consists of about 25 000 lines of code and is written in FreshOcaml, a variant of INRIA's Ocaml with support for automatic alpha conversion. It is expected to be released in January 2005.

### 4.2 Language Design

This work addresses the design of distributed languages. Our focus is on the higher-order, typed, call-by-value programming of the ML tradition: we concentrate on what must be added to ML-like languages to support typed distributed programming. We explore the design space and define and implement a programming language, ACUTE, [P10].

This builds on previous work done by James J. Leifer, Gilles Peskine (INRIA), Peter Sewell, Keith Wansbrough (U. of Cambridge), published in ICFP 2003. The present work extends the theory to contend with the challenge of integrating with an ML like programming language. This requires a synthesis of novel and existing features.

**Type-safe marshalling** Type-safe marshalling demands a notion of type identity that makes sense across multiple versions of differing programs. For concrete types this is conceptually straightforward, but with abstract types more care is necessary. We generate globally-meaningful type names either by hashing module definitions, taking their dependencies into account; freshly at compile-time; or freshly at run-time. The first two enable different builds or different programs

to share abstract type names, by sharing their module source code or object code respectively; the last is needed to protect the invariants of modules with effect-full initialisation.

**Dynamic linking and rebinding** Dynamic linking and rebinding to local resources in the setting of a language with an ML-like second-class module system raises many questions: of how to specify which resources should be shipped with a marshalled value and which dynamically rebound; what evaluation strategy to use; when rebinding takes effect; and what to rebind to. For ACUTE we make interim choices, reasonably simple and sufficient to bring out the typing and versioning issues involved in rebinding, which here is at the granularity of module identifiers. A running ACUTE program consists roughly of a sequence of module definitions (of ML structures), imports of modules with specified signatures, which may or may not be linked, and marks which indicate where rebinding can take effect; together with running processes and a shared store.

**Global expression names** Globally-meaningful expression-level names are needed for type-safe interaction, e.g. for communication channel names or RPC handles. They can also be constructed as hashes or created fresh at compile time or run time; we show how these support several important idioms. The polytypic support and swap operations of Shinwell, Pitts and Gabbay's FreshOCaml are included to support renaming of local names occurring inside of values during communication.

**Versioning** In a single-program development process one ensures the executable is built from a coherent set of versions of its modules by controlling static linking often by building from a single source tree. With dynamic linking and rebinding more support is required: we add versions and version constraints to modules and imports respectively. Allowing these to refer to module names gives flexibility over whether code consumers or producers have control.

**Local concurrency** Local concurrency is important for distributed programming. ACUTE provides a minimal level of support, with threads, mutexes and condition variables. Local messaging libraries can be coded up using these, though in a production implementation they might be built-in for performance. We also provide thunkification, allowing a collection of threads (and mutexes and condition variables) to be captured as a thunk that can then be marshalled and communicated (or stored); this enables various constructs for mobility to be coded up.

We deal with the interplay among these features and the core, in particular with the subtle interplay between versions, modules, imports, and type identity, requiring additional structure in modules and imports. We develop a semantic definition that tracks abstraction boundaries, global names, and hashes throughout compilation and execution, but which still admits an efficient implementation strategy.

The definition is too large to make proofs of the properties feasible with the available resources and tools. To increase confidence in both semantics and implementation, therefore, our implementation can optionally type-check the entire configuration after each reduction step. Our strategy has been to synchronise the development of the formal specification of the ACUTE language with that of its implementation. As a result we have been able to quickly discover and fix errors in the type-system and in the run-time semantics.

The specification of ACUTE, together with a discussion of the design rationale, is available as an INRIA Research Report [P10].

## 5 Mozart/Oz

### 5.1 P2PS

The P2PS library implements the Tango protocol, which was published in EuroPar 2004 (August 2004). P2PS itself is published as a software package in the MOGUL archive of the MOZART Programming System [s3]. The P2PS library is described in a publication in the MOZ 2004 conference [P7]. Tango [P1] is a variant of DKS [3] that improves scalability (bigger size networks are possible with the same size routing tables) by decreasing the path redundancy (the number of paths between two nodes, which is exponential in DKS). The correction-on-use technique of DKS is directly applicable to P2PS.

During 2004 we have robustified and extended P2PS. P2PS now contains a reliable send primitive: the send is guaranteed to arrive if the source and destination nodes are working and if the topology maintenance is working correctly. We have ported P2PS to the PlanetLab infrastructure [1] and are running it on 100 nodes. PlanetLab is a highly dynamic worldwide infrastructure that is sponsored by Intel. Because it is based on a set of machines and the goodwill of network administrators at hundreds of institutions, it has frequent failures and network problems. Successfully running P2PS on this infrastructure is a good test of robustness in a highly dynamic environment.

In collaboration with INRIA, we have implemented an application on top of P2PS called Post-It. This application is explained in the deliverable D5.3.

### 5.2 P2PKit

We are now working on a service architecture for P2PS, called P2PKIT [P3]. This architecture allows to install, monitor, and update peer-to-peer services, where a *service* is a set of component instances that collaborate to provide functionality to other component instances. At the end of 2004, we have a running prototype of this service architecture that we will continue to develop in 2005.

P2PKIT is a programmer's toolkit for Distributed Hash Table (DHT) based peer-to-peer middleware systems such as Chord, CAN, P2PS, Pastry and Tapestry. Our implementation is written in Oz [s3] and built on top of the P2PS library. It is provided as a set of ozmake packages for easy installation. The programmer's model of P2PKIT is presented in the deliverable D3.7 [2].

We are using P2PKIT to investigate and implement robust, decentralised services for replication, data storage and retrieval, distributed transactions, and so on. We are also writing applications, e.g., a graphical Control Center, and deploying them on wide-scale distributed networks such as PlanetLab [1] and Evergrow [5].

Four final year students at UCL will be using P2PKIT to develop content storage services and a secure, peer-to-peer, multi-user dungeon (MUD) system.

### 5.3 Fault tolerance and the DSS

During the development of P2PS, we have noticed that the fault detection primitives of the MOZART system are often complex to use. We have therefore designed a simpler fault detection mechanism. The basic idea is that a distributed language entity exists on each process as a full-fledged entity, with a synchronization protocol between the processes. If there is a failure, then the synchronization protocol is replaced by a simpler protocol or by no protocol at all. Experience shows that this mechanism is much easier to program with than the existing mechanism. In particular, separation of concerns is supported better. The new proposal was published in MOZ 2004.

## **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.

### **6.1 JDSS**

A description of the implementation details is available in deliverable D3.8 [4].

We mainly focused our efforts on the following tasks:

- Adapting the JDSS to the current DSS version. During 2004 we have continued our efforts to adapt the JDSS to the CVS version of the DSS.
- Testing the JDSS with test examples written in Java/Scala. The functionality provided by the JDSS needs more extended testing. The JDSS software is still in alpha stage and would need further work before demonstrator applications can be build on it. While implementing our test examples in Java and in C we encountered several difficulties in dealing the different semantics of the Java and C communication libraries.
- Integrating the JDSS into the DSS software repository; this includes the automatic makefile generation.

## 7 Dissemination

### 7.1 Publications

- [P1] B. Carton and V. Mesaros, *Improving the Scalability of Logarithmic-Degree DHT-based Peer-to-Peer Networks*, Euro-Par 2004, Pisa, Italy, [http://www.mozart-oz.org/mogul/doc/cetic\\_ucl/p2ps/pepito/Tango.pdf](http://www.mozart-oz.org/mogul/doc/cetic_ucl/p2ps/pepito/Tango.pdf)
- [P2] B. Carton and V. Mesaros, *P2PS Library*, Mozart Consortium 2004, [http://www.mozart-oz.org/mogul/doc/cetic\\_ucl/p2ps/p2ps/](http://www.mozart-oz.org/mogul/doc/cetic_ucl/p2ps/p2ps/)
- [P3] Kevin Glynn, *P2PKit: A Programmer's Toolkit for P2PS*, Université Catholique de Louvain, 2004, <http://renoir.info.ucl.ac.be/twiki/pub/INGI/P2PKit/lightp2pkit.pdf>
- [P4] D. Grolaux, K. Glynn, P. van Roy, *A Fault Tolerant Abstraction for Transparent Distributed Programming*, MOZ 2004, 2nd International Mozart/Oz Conference, Charleroi, Belgium, <http://www.cetic.be/moz2004/talks/FaultModel.ppt>
- [P5] D. Havelka, Ch. Schulte, P. Brand and S. Haridi, *Thread-based Mobility in Oz*, MOZ 2004, 2nd International Mozart/Oz Conference, Charleroi, Belgium (will appear in the LNAI series, Springer-Verlag)
- [P6] James Leifer, Michael Norrish, Peter Sewell and Keith Wansbrough, *Acute and TCP: Specifying and Developing Abstractions for Global Computation*, Proceedings of the APPSEM II Workshop, Tallinn, April 2004, 2ff.
- [P7] V. Mesaros, B. Carton, and P. van Roy, *P2PS: Peer-to-Peer Development Platform for Mozart*, MOZ 2004, 2nd International Mozart/Oz Conference, Charleroi, Belgium, [http://www.cetic.be/moz2004/talks/p2ps\\_moz2004.ppt](http://www.cetic.be/moz2004/talks/p2ps_moz2004.ppt)
- [P8] Martin Odersky and al., *An Overview of the Scala Programming Language*, Technical Report IC/2004/64, EPFL Lausanne, <http://icwww.epfl.ch/publications/list.php>
- [P9] Martin Odersky and Matthias Zenger, *Independently Extensible Solutions to the Expression Problem*, Proc. FOOL 12, January 2005, <http://homepages.inf.ed.ac.uk/wadler/fool/>
- [P10] Peter Sewell and al., *Acute: high-level programming language design for distributed computation. Design rationale and language definition*, University of Cambridge Computer Laboratory, 2004, Technical Report 605, and INRIA RR-5329. page 193ff.
- [11] Gareth Stoye, Mike Hicks, Gavin Bierman and Iulian Naemtiu, *Mutatis Mutandis: Safe and Predictable Dynamic Software Updating*, Proceedings of POPL 2005, January 2005 (to appear).

## 7.2 Web sites

- [s1] Peter Sewel and al., *Acute*, INRIA/UCAM, 2004,  
<http://www.cl.cam.ac.uk/~pes20/acute/>
- [s2] SICS, *The General Distributed SubSystem*, SICS, 2004,  
<http://dss.sics.se/>
- [s3] Mozart Consortium, *The Mozart Programming System*, UdS, SICS, UCL, 2004,  
<http://www.mozart-oz.org/>
- [s4] Rachid Gerraoui and al., *The DACE Project*, EPFL. 2004,  
<http://www.d-a-c-e.com/>
- [s5] Martin Odersky and al., *The Scala Programming Language*, EPFL, 2004,  
<http://scala.epfl.ch/>
- [s6] Bruno Carton and Valentin Mesaros, *P2PS: A Peer-to-Peer Library Offering Primitives and Services for Applications*, *Software Library*, MOGUL Archive, Mozart Programming System, 2004, <http://www.mozart-oz.org/mogul/>
- [s7] Kevin Glynn, *P2PKit*, UCL, 2004  
<http://renoir.info.ucl.ac.be/twiki/bin/view/INGI/P2PKit>

## References

- [1] PlanetLab Consortium. PlanetLab: An open platform for developing, deploying, and accessing planetary-scale services, 2004. <http://www.planet-lab.org/>.
- [2] Kevin Glynn. *Extending the Oz language for peer-to-peer computing*, 2004. <http://www.sics.se/pepito/deliverables.html>.
- [3] P. Brand L. Onana, S. El-Ansary and S. Haridi. DKS(N,K,f): A family of low communication, scalable and fault-tolerant infrastructures for P2P applications. In *Proc. of CCGRID*, May 2003.
- [4] 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>.
- [5] Evergrow Project. EVERGROW - Building the Internet of 2025, 2004. <http://www.evergrow.org/>.