



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

**Deliverable no: D3.6**

## **Peer-to-peer Oz platform (2)**

REPORT VERSION: first

REPORT PREPARATION DATE: 2004.06.30

CLASSIFICATION: Public

DELIVERABLE NO: D3.6      DUE DATE: Month 30      DELIVERY DATE: Month 30

PROJECT START DATE: 2002.01.01      PROJECT DURATION: 36 months

RESPONSIBLE PARTNER: UCL

PARTICIPATING PARTNERS: UCL, SICS

PROJECT COORDINATOR: Swedish Institute of Computer Science AB

PROJECT PARTNERS: EPFL Lausanne, INRIA Paris, KTH Stockholm, UCL Louvain, University of Cambridge UK



**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: D3.6  
Peer-to-peer Oz platform (2)  
Due date: project month 30  
Delivery date: 2004-06-30

Responsible Partner: UCL  
Participating Partners: UCL, SICS

30th June 2004

Valentin Mesaros, Bruno Carton, Kevin Glynn, and Peter Van Roy

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Relation to other workpackages</b>	<b>3</b>
<b>3</b>	<b>Structured P2P systems: overview</b>	<b>4</b>
<b>4</b>	<b>Functionality</b>	<b>4</b>
4.1	Create a network . . . . .	5
4.2	Join a network . . . . .	5
4.3	Leave a network . . . . .	5
4.4	Message exchange . . . . .	6
4.5	Monitoring . . . . .	6
4.6	Others . . . . .	6
<b>5</b>	<b>Architecture</b>	<b>7</b>
5.1	COM layer . . . . .	7
5.2	Core layer . . . . .	8
5.3	Services layer . . . . .	9
<b>6</b>	<b>Reliable communication primitive</b>	<b>9</b>
6.1	Implicit connectivity . . . . .	10
6.2	Related API . . . . .	10
<b>7</b>	<b>A simple example using P2PS</b>	<b>11</b>
7.1	Three peer system . . . . .	12
7.2	Dictionary functionality . . . . .	13
<b>8</b>	<b>Applications using P2PS</b>	<b>14</b>
8.1	Community Panel . . . . .	14
8.2	PostIt . . . . .	15
<b>9</b>	<b>Conclusion</b>	<b>16</b>
<b>10</b>	<b>Further work</b>	<b>17</b>

## 1 Introduction

As stated in Annex 1 of the PEPITO project, one of the main objectives of WP3 is to extend Oz and Java to reflect new programming abstractions that use the protocols of workpackage 2 (WP2), in order to provide these languages with peer-to-peer (P2P) abilities. This deliverable is focused on extending Oz-Mozart [2] and it is the second report on presenting the progress in WP3 with respect to the development of the peer-to-peer Oz platform, called P2PS [3]. P2PS provides the developer with the possibility of building and working with P2P overlay applications, offering different P2P primitives and services. Although independent of the underlying P2P technology, P2PS currently only supports Tango [7, 8]. Tango is a peer-to-peer algorithm, that we developed in WP2, to better structure relative exponential networks to increase their scalability. Note that the Tango algorithm was previously (in deliverable D3.5) called GSChord.

Since the first report on “D3.5: *Peer-to-peer Oz platform (1)*”, the P2PS peer-to-peer platform has been enriched with additional functionality and features. First, we should say that P2PS has a new API which is more flexible and more ergonomic than the previous one. Indeed, given the feed-back we had from developers using P2PS, we found it very useful to improve its API. The complete API is provided as an attached document to the present deliverable. Second, we should mention the fact that the functionality of P2PS has been enriched with one-to-one reliable communication primitive, and that work on providing one-to-many communication primitive is currently undertaking. Finally, we should say that for P2PS we took the choice of carrying out with the Tango peer-to-peer algorithm and temporarily stopping the support for DKS [6]. We took this decision because DKS and Tango are very similar, but still the functionality of Tango was presented to be more attractive. However, given the modularity of P2PS, support for DKS can be easily provided with quite small effort. Moreover, the new API is designed to support both Tango and DKS algorithms.

P2PS is delivered as a software package [3] containing the source code together with an API documentation and an example-based user tutorial. It is ready to be used both, by researchers to develop and test new algorithms, and by programmers to develop P2P applications. On the 30<sup>th</sup> of October 2003 we made a public release of P2PS on MOGUL [1]; the official archive of Oz-Mozart libraries. Since then, P2PS has become known to researchers in the domain of overlay networks and P2P systems and its web site is daily visited.

The remainder of this document is organized as follows. We continue with the relation of our work to other workpackages. In Section 3 we briefly recall the principles of structured P2P systems. In Section 4 we describe the main functionality provided by P2PS. In Section 5 we describe the architecture of P2PS, while in Section 6 we describe the internals of the communication primitive. In Section 7 we show how to write a simple application for P2PS and how to extend it, and in Section 8 we describe two existing applications that use P2PS. Then, we conclude and describe how we see P2PS is going to evolve from now on.

## 2 Relation to other workpackages

The work in this deliverable is related to other workpackages in PEPITO as follows:

**WP2** The goal of workpackage 2 is to develop fully decentralized, scalable, fault-tolerant and self-organizing application-independent infrastructure that ease the development of various P2P platforms and applications. Most of the algorithms proposed in WP2 will be implemented by the P2PS library and tested with different applications. Thus, implementing and testing the algorithms in Oz-Mozart, provides us with quick results that can be used as feed-back to WP2.

**WP4** The distribution language-independent middleware component (i.e., DSS) being developed in WP4 will be extended with P2P services. Furthermore, one of the main objectives of WP3 is to integrate P2P abilities in Oz-Mozart. Since P2PS offers P2P primitives, providing a first Oz implementation of different algorithms proposed in WP2, the API proposed by P2PS represents a first attempt towards what can become the peer-to-peer API in Oz language as well as for the peer-to-peer functionality of DSS.

**WP5** Workpackage 5 is in charge with building applications demonstrating the peer-to-peer functionality of the systems developed in the project. The P2PS library is available to be (and as described in Section 8, it is already) used for developing P2P applications in Oz. On the other hand, by using the P2PS library with realistic applications, allows us to continuously improve its API, based on the feed back we get from application developers.

### 3 Structured P2P systems: overview

In this section we briefly recall the principles and notations of the structured P2P networks. We take the Chord algorithm [12] as a case study since a) Chord is one of the first P2P algorithms based on the idea of Distributed Hash Table – DHT, and b) Tango and Chord have many commonalities.

A first characteristic of a structured P2P system is that it is DHT-based, where key#value pairs are associated to nodes in the overlay network depending on the “distance” between the key *id* and the nodes’ *ids*. (Hereinafter, we will use the term *node* to refer to both the node and its identifier under the hash function, as the meaning will be clear from the context.). Both, nodes and keys, take values in the same identifier space. In the case of Chord, the identifier space is a virtual ring within which hashed node and data item key *identifiers* are spread by using a consistent hashing.

An other important characteristic of a structured P2P system is the fact that the overlay network is well defined in order to achieve logarithmic key lookup. With the advent of the DHT-based systems, the main procedure in the P2P systems, the key *lookup*, was provided with some guarantees. For instance, a lookup for a key will not take more than a certain maximum number of hops.

In Chord each node has a *predecessor* and a *successor* representing references to the previous and respectively the subsequent node in the identifier space. A key is stored at the node succeeding the *id* of that key on the circular identifier space. Thus, the naive lookup procedure for a certain key reduces to looking for the first node whose *id* is greater than, or equal to the *id* of that key along the identifier space, going clockwise. To speed up the lookup process, each node maintains supplementary references (called *fingers*) about some other nodes inside a *routing table*. Given an identifier space of size  $N = 2^k$ , beside the references to its predecessor and successor, each node in the Chord system stores  $k$  fingers. Note that in structured P2P systems there is a tradeoff between the size of the routing table at each node and the maximum number of hops a request would take when looking for a key.

### 4 Functionality

In this section we present the main functionality provided by our P2P platform called P2PS: *Peer-to-Peer System*. This functionality is provided via the class `P2PS.p2pServices`. The P2PS library provides the developer with the possibility of building and working with P2P overlay applications, offering different P2P primitives and services. P2PS is providing the distributed peer-to-peer applications with a means to organize themselves in large scale structured overlay networks as well as

providing them with management and communication primitives whose costs evolve logarithmically with the system size. For details on its API you can refer to the attached document to this report.

The main functionality provided by the P2PS library can be summarized as follows: network management primitives such as create, join and leave a network, communication primitives such as one-to-one, broadcast and multicast, and monitoring primitives. With P2PS we intended to provide basic P2P primitives on top of which more specialized services to be built. Looking for the responsible of a key is not provided as a basic primitive in P2PS. Instead, the main basic primitives are sending and receiving a message from one node to another. Nevertheless, dictionary operations can be immediately provided by using the communication primitives offered by the P2PS library (see Section 7.2). Furthermore, we have undergoing research to extend the functionality of P2PS with other primitives (see Section 10).

#### 4.1 Create a network

This functionality provides the programmer with the possibility to create a P2P overlay network. It will create the first node of a network. What this actually means is that an `AccessPoint` is created for this node (for the description of the access point, see Section 5.1). In order to create a network in P2PS, one will use the method `createNet`. This method can be featured with different overlay network and node parameters (e.g., the maximum overlay network size, the *id* of this node), as well as with parameters related to the local access point (e.g., IP and port number). Then, after creating a peer node, its access point can be published, thus allowing remote connections to it from other nodes. Furthermore, the node is provided with message and event streams on which messages from other nodes and respectively different node and network events would eventually be accessible.

#### 4.2 Join a network

When joining an overlay network, a peer node  $n$  needs to have the knowledge of an `AccessPoint` of another peer node  $p$  already present in the respective overlay network. The underneath protocols will actually join  $n$  to the network via the node  $p$ . Note that node  $p$  serves only as an entry point to the network for node  $n$ . Generally, the position of a joining node into the system does not depend on the entry point it uses to get into the network. In order to join a network in P2PS, one will use the method `joinNet`. This method can be featured with different node parameters (e.g., the *id* of this node), as well as with parameters related to the local access point (e.g., IP and port number). As any other node in the overlay network, a new joined node will be associated an access point. Furthermore, once inside the network, a node may receive messages from other nodes from the network, and node and network events on the associated message and respectively event streams.

#### 4.3 Leave a network

Leaving an overlay network means implicitly disconnecting this node from all the other nodes it is connecting to in the overlay network. Although, generally, a P2P network tolerates node failures, it is expected that a node makes a gracefully leave. Thus, underneath, a node will run a simple protocol to disconnect from its neighbors. In order to leave a network in P2PS, one will use the method `leaveNet`. This will terminate the message and the event streams.

## 4.4 Message exchange

P2PS provides end-to-end communication primitives. That is, exchanging messages between peers throughout the overlay network. Due to its organization, the system can perform efficient key based routing. That is, a message from a node  $s$  to a node  $d$  is routed throughout the overlay network according with the corresponding key lookup procedure, where  $d$  is considered a key.

Note that one can indicate to send a message either to the node responsible for the key with value  $d$ , or directly to the node whose  $id$  equals  $d$ . While in the former case the message may eventually always reach its destination, in the latter the destination may simply not be present. Both flavors of the message delivery are useful in practice. To send one-to-one messages in P2PS, one will use the method `send`. To send reliable one-to-one messages, one will use the method `rSend`. For details on the communication reliability, see Section 6.

Another communication primitive provided by P2PS is one-to-many. Simple `broadcast` and `multicast` is provided. Both protocols employed are based on an idea [9] developed in WP2. That is, exploit the tree structure of a Chord-like systems. In the case of the broadcast, the message is sent to all the nodes in the network. In the case of multicast, the message is sent to a given list of nodes; that is, explicit multicast. As in the case of one-to-one messages, in the case of multicast one can choose to send the message either to nodes' responsables, or directly to the nodes whose  $ids$  equal those in the destination list.

To increase its reliability, an application might decide to replicate the content stored at a node to some of the node's successors (i.e., the successor list). The method `sendToSucc` can be used to send a message to a number of successors of a node (whoever they be).

## 4.5 Monitoring

In a dynamic network as P2P overlay networks, being aware of the status and of the changes with respect to the peer node and the network might be very useful for the upperlying application. A good example is the application running on nodes with limited resources. Thus, in P2PS we decided to provide a set of events on the event stream associated with each peer node. These events indicate changes on the connections with the node's predecessor and successors. This way, for example, if the successor of a node has changed, the application can do replication on the new successor.

Another way of monitoring the peer node is offered by the method `getStatistics`. It provides a set of information – most of it in the form of counters – about the status of the peer node. For example, one can have information about the followings: the number of incoming and outgoing connections, the number of data and control messages sent by this node, the number of data and control messages forwarded by this node.

## 4.6 Others

### Deal with asymmetric connections

In order to guarantee the system correctness (i.e., each node is reachable using the routing algorithm), the overlay has to organize itself in to a virtual ring such that each node points to the right successor. However, in the case of asymmetric connection (e.g., caused by a firewall) between a node  $n$  and its successor  $p$ , this is not possible. The solution that we provide in P2PS has a proxy-based approach. The node to which symmetric connections can not be established (let's call it *hidden node*) will have only limited functionality within the system. It will use its successor as a proxy to get access to the system.

A hidden node differs from an ordinary node in the following matters. First, although it keeps an up-to-date routing table, a hidden node is not referred to as a finger by any other node in the system. Second, it does not participate in the routing procedure. Finally, it is only responsible of the key corresponding to its *id*. Moreover, the proxy node keeps a reference to the hidden nodes pointing to it in order to deliver them the messages their are addressed.

From the point of view of the programmer, the differences between a hidden node and an ordinary node are perceived relatively to two events: in the event `connected` the feature `useProxy` equals `true`, and the event `newpred` does not occur at all (since the hidden node does not have a predecessor). Additionally, the hidden node is addressed only the messages with the same *id* as his. Note that, when joining the overlay network, a peer node by itself can implicitly choose to act, or not, as a hidden node.

### Low cost control

In order to route messages, each peer maintains a routing table which partially reflects the network structure. Thus, each time the network topology is modified, the routing table entries must be updated. For this end, in P2PS we use the *correction-on-use* mechanism – proposed in [6] and developed in WP2 – which is based on the network usage. Indeed, instead of probing the network periodically to detect topology modifications, a peer is notified about an update when it forwards a message to a “wrong” peer. This mechanism saves network bandwidth since it does not produce any extra messages for an unused network.

## 5 Architecture

The P2PS library is organized in three layers: **COM**, **Core**, and **Services** (see Figure 1). They correspond to P2P services provided to the application: structural operations in order to preserve overlay network properties, and message exchange and channel establishment operations.

### 5.1 COM layer

The **COM** layer is in charge with interfacing with the underlying physical network. It provides functionality to the **Core** layer. Basically, **COM** will provide communication functionality through a common API, regardless the physical protocol employed, such as TCP, UDP, and Bluetooth. The functionality provided by the **COM** layer is: access point creation, connection establishment, basic communication primitives, and fault detection.

**Access point creation.** We define an access point to be an addressable entry point to a node. It is the **COM** layer who defines the form and the meaning of an access point. Moreover, the representation of an access point – `AccessPoint` will have a meaning only to the **COM** layer. It can, for example, be defined as an `ipAddr/socketNr` pair, but its definition can also be security-flavored. The access point creation primitive consists in creating an access point for a peer node. There can be one access point per peer node. Then, a node peer can publish its access point, thus allowing remote connections to it.

**Connection establishment.** The connection establishment offers `connect` and `disconnect` primitives for point-to-point connecting to and respectively disconnecting from a node, given its `AccessPoint`.

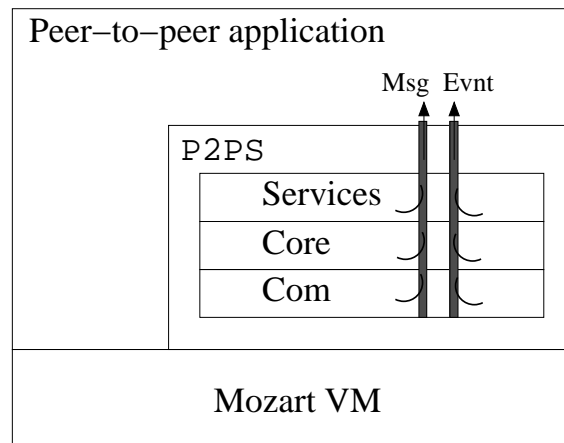


Figure 1: The three-layer architecture of a P2PS node, and its interaction with the application and the transport module (provided by Mozart VM, here).

**Basic communication primitives.** The basic communication primitives provided are `send message` and `receive message`. They are point-to-point primitives providing reliable data transfer over a connection established via an `AccessPoint`. For this end, P2PS uses the `Sockets` [5] and `Session` [4] modules developed at UCL on top of Mozart.

**Fault detection.** The fault detection primitives provide with a means for detecting different types of network anomalies relative to point-to-point connection such as permanent faults and temporary faults as well as announcing them to higher levels. For this end, again, P2PS uses the `Sockets` and `Session` modules. Given the high dynamics of a P2P network, this functionality is very important.

## 5.2 Core layer

An overlay network topology can be viewed as a graph composed of arcs and nodes. The purpose of the `Core` layer is to provide high-level connectivity primitives between peer nodes, thus allowing to add and remove arcs to and respectively from a node.

The `Core` layer, as its name indicates, is the central component of the P2PS library. It implements the `Tango` [7, 8] algorithm. Its purpose is threefold: implement node join and leave mechanisms, route key based messages to their responsible, and maintain the routing table and the successor list regardless the nodes joining and leaving.

**Joining/leaving a network.** Given an entry (i.e., an `AccessPoint`) to the system, the join mechanism consists in finding the right place of the joining node within the overlay network, i.e., between its successor and predecessor, and establishing a communication channel with them. Obviously, the predecessor and successor of the joining node are affected by the join operation and therefore they must update their references in order to reflect the network change. The particularity of the implemented distributed join is the fact that it is atomic. Indeed, once the joining node  $n$  has located its successor  $p$ , it asks  $p$  to insert it into the system. If a node receives an insertion request while inserting another node, it will delay the request until the current insertion has finished. Furthermore, a node can perform an insertion only once correctly inserted

into the overlay network. The leave operation is much simpler and consists only in advertising its connected peers about the leave, and disconnecting from them.

**Routing messages.** The message routing algorithm is based on what is called the *key lookup* primitive. It consists in handing the incoming message to the upper layer if it reached its destination (i.e., if the receiver peer is responsible of the message identifier) or forwarding the message to the closest peer entry of the routing table, according to the routing metric used.

**Topology maintenance.** Another operation is overlay topology maintenance, or routing table maintenance. This procedure is run at each node and consists in maintaining connections to well defined neighbors in order to ensure certain global guarantees (e.g., a lookup for a key will not take more than a certain maximum number of hops). Instead of correcting the routing table explicitly by probing periodically the neighbors, the routing table in P2PS is corrected implicitly when the peers are actually using the network (as described in [6]). While this economic way is well suited for maintaining the routing table, it is absolutely not for maintaining the successor list of a node. Since the reason to keep the successor list is to preserve the network coherence (i.e., when the successor of a peer failed, the peer has to refer to the next peer in the successor list), a peer should be notified about all modifications of the  $r$  next succeeding peers ( $r$  stands for the length of the successor list).

### 5.3 Services layer

The **Services** layer is a kind of wrapper, building up the raw primitives offered by the **Core** layer; operations needed to implement peer-to-peer applications. These operations can fit into two categories. First is the overlay network management which comprises system initialization, create connection access, and system join and leave operations. Second are the communication primitives at the overlay network level which comprise one-to-one message send, and message broadcast and multicast operations. Note that in the current version of P2PS the reliable one-to-one communication primitive is provided (for more details on communication reliability refer to Section 6).

The application can interact with the **Services** layer by invoking the required methods directly and by simply reading information on the two available streams: `message` and `event` associated with each peer node. The message and event streams are a way of asynchronously obtaining information about the received messages and respectively the node and network events.

## 6 Reliable communication primitive

In many situations, one would like to have reliability communication between nodes within a network. Moreover, it would be very useful to have different degrees of reliability (e.g., FIFO order, partial order). As in any network, in an overlay network, in order to provide reliability communication between any two nodes within the network, it does not suffice to ensure the communication reliability on all the edges within that network. The reason for this is that, usually, a connection spans over more than two nodes, and these nodes have a certain probability to fail. Since most P2P systems are dynamic (i.e., relatively high rate of nodes joining/leaving) providing reliable communication primitives is be very useful.

In our previous report, D3.5, on P2PS we mentioned the fact that the P2PS library provided reliable message sending, except in the situation where nodes on the path between sender and receiver disconnect during the message send, and thus losing that message. Since then, work has been

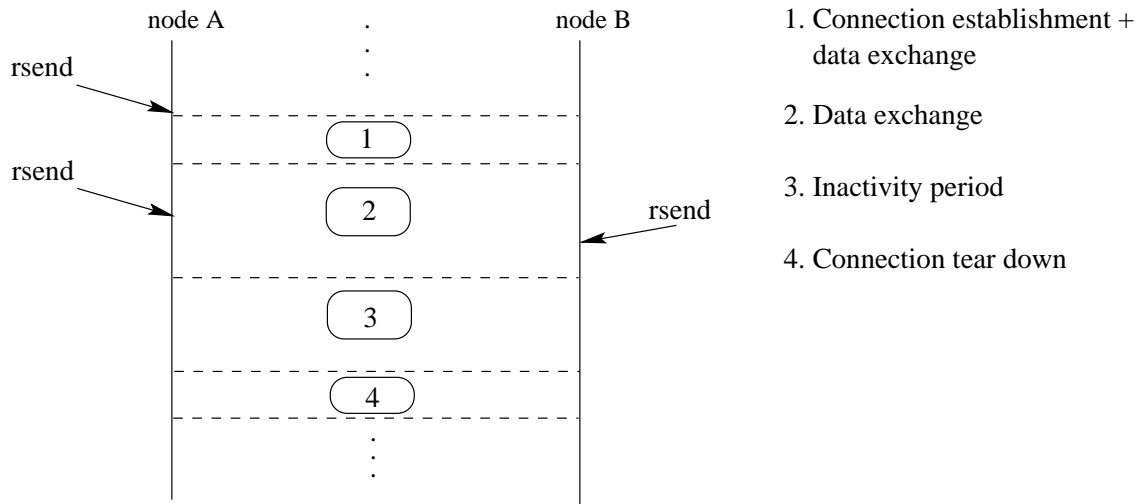


Figure 2: Different states of a connection between two nodes. Node A initiates connection to node B.

conducted to introduce reliable communication primitive for P2PS and thus achieving communication reliability even in the face of frequent node failures. Thus, we introduced reliable one-to-one communication into P2PS. Moreover, we are undertaking research and work for introducing reliable one-to-many communication. We continue this section with describing the design decisions we took when introducing the communication reliability into P2PS.

## 6.1 Implicit connectivity

In order to provide reliable communication between two nodes in a network, a connection has to be established between them. This implies sharing a state – consisting of messages sent, received, or lost – at both nodes. When designing the connection protocol we followed the idea of *end-to-end arguments* [11], where it is suggested that only the nodes involved in the connection should share any connection state. Furthermore, we don't assume routing symmetry for the nodes sharing a connection. That is, the paths (in both directions) between the nodes do not have to be the same.

One of our main goals is to provide reliability through a simple API. Thus, we decided to use implicit connectivity. That is, when a reliable send is invoked from a node A to a node B, unless already existing, a new connection is established between the two nodes. In this case, the first of the two nodes invoking the reliable send will initiate the connection. After the connection established, all messages exchanged between the two nodes will obey the reliability rules of that connection, e.g., all messages will eventually be delivered in order. Furthermore, after an inactivity period, a connection between two nodes can be torn down by any of the parties. Again, this is done implicitly. In Figure 2 we show the states of a connection between two nodes A and B, with node A initiating the connection.

## 6.2 Related API

In this subsection we describe the reliable communication method within P2PS, adding different functionalities to it, step by step. For the moment, the basic one-to-one send together with the synchronous send functionality is already implemented. We have ongoing work on providing functionality on send to responsible and reliability degree as well as on reliable multicast.

**One-to-one send.** For the programmer, simply sending a reliable message from one node peer to another should not be much different than the best-effort `send` method. The following method invoked at a node `A` will open a connection, unless it already exists, between node `A` and node `B` and deliver the message `Msg` to `B`. The send is reliable in the sense that all messages will eventually be delivered in a FIFO manner.

```
rsend(dst:B msg:Msg)
```

**Synchronous and asynchronous send.** The communication procedures in P2PS were asynchronous. With the introduction of communication reliability, the communication procedures can be made synchronous, if need be. This can be done with the optional parameter `Ack`, and by taking advantage of the *data-driven* mechanism in Oz-Mozart. Indeed, the `Ack` variable will be bound either to `acked` as soon as it is known that the message was acknowledged as received, or to `nonacked` when the connection is closed and there has not been received any acknowledgment for that message.

```
rsend(dst:Nid msg:Msg ack:Ack)
```

**Send to responsible.** The communication primitives in P2PS provide the application with the possibility of sending messages not only to a node peer identified by a node id `NID`, but also to the node (who ever it is) responsible for a given key. This can be done with the optional parameter `TR` which is a boolean specifying whether, or not, to send the message to the responsible of `NID` (here, `NID` is considered to be a key). If during the connection the responsible of the key has changed, the application is notified.

```
rsend(dst:Nid msg:Msg ack:Ack toResp:TR)
```

**Message context and reliability degree.** In order to easily recover the non acked messages when the connection is ungracefully closed, a `context` can be provided with each message to be sent. This context, that can be any Oz entity (e.g., integer, procedure), will be paired with each non acked message returned by P2PS with the event `closeConn`.

```
rsend(dst:Nid msg:Msg ack:Ack toResp:TR context:C rdegree:RD)
```

**Connection events.** The fact of opening a connection to a node `PID` is notified to the programmer by the event `openConn`. Afterwards, closing a connection is notified to the programmer by the event `closeConn`. Both events are raised at both nodes involved with the connection. The remaining non acked messages (each message paired with the corresponding `context`) are available in the list `RM`. These two are additional events to the event set that P2PS provides to the application via the event stream associated with each peer node.

```
openConn(peerId:PID)
```

```
closeConn(peerId:PID rmnMsgs:RM)
```

## 7 A simple example using P2PS

We present a simple example of a P2P system composed of three peers which uses the P2PS library. The system is composed of three nodes, `node1`, `node2`, and `node3`, where `node2` and `node3` join the system through `node1` and respectively `node2`. In this example `node3` sends a multicast

message to `node1` and `node2`, and an one-to-one message to `node1`. Furthermore, we extend this example with Dictionary functionality. The code runs directly in the Oz Programming Interface of Mozart. Since P2PS is implemented in Oz-Mozart, one needs the Oz virtual machine to compile and use it.

## 7.1 Three peer system

The first node of a P2P system is always special. Actually, it represents a system by itself. In our case, `node1` decides to work on port number 3001 and take the `nodeId` equal to 1. It runs a loop over the message stream and displays the messages addressed to it. The following is the code implementing `node1`.

```
declare /* node1 */
[P2PS] = {Module.link ['x-ozlib://cetic_ucl/p2ps/P2PS.ozf']}
MS

% Create first node (with id 1) in a P2PS network.
% Set the port# in the access point config to 3001.
OP2PS = {New P2PS.p2pServices
         createNet(nodeConfig: nodeConfig(nodeId:1)
                  apConfig:   apConfig(pn:3001)
                  msgStrm:   MS)}

% Display each message received on the message stream.
for M in MS do
  {Show M}
end
```

Then, we create `node2` with `nodeId` 16. This node joins the system via `node1` specifying its remote `AccessPoint` as the IP address and port number. Further on, we run a loop to wait and show the messages sent to this node. The following is the code implementing `node2`.

```
declare /* node 2 */
[P2PS] = {Module.link ['x-ozlib://cetic_ucl/p2ps/P2PS.ozf']}

% Build the access point token of a node (node1) in the P2PS network.
RAP = {P2PS.address2ap "127.0.0.1" 3001}

% Create a node with id 16 and join the network, using the token RAP.
OP2PS = {New P2PS.p2pServices
         joinNet(remoteAP:   RAP
                nodeConfig: nodeConfig(nodeId:16)
                apConfig:   apConfig(pn:3002))}

% Get the message stream and display each received message.
for M in {OP2PS getMsgStrm($)} do
  {Show M}
end
```

Finally, we create `node3` without specifying its `nodeId`; the node will be provided with a random `id`. This node chooses to join the system via `node2`, specifying its address and port number.

Note that it could have chosen to join via any other node within the network, e.g., node1. Further on, it sends a reliable one-to-one message to node1, and a multicast message to node1 and node2. The following is the code implementing node3.

```
declare /* node3 */
[P2PS] = {Module.link [^x-ozlib://cetic_ucl/p2ps/P2PS.ozf^]}

% Build the access point token of a node (here, node2) in the P2PS network.
RAP = {P2PS.address2ap "127.0.0.1" 3002}

% Create a node and join the network, using the token RAP.
OP2PS = {New P2PS.p2pServices joinNet(remoteAP: RAP)}

% Reliably send a message to node with id 1.
{OP2PS rsend(dst:1 msg:sasa)}

% Send a multicast message to nodes with id 1 and 16.
{OP2PS multicast(dst:[1 16] msg:hello)}
```

## 7.2 Dictionary functionality

The previous example shows how to construct a P2P system and how the P2PS library can be used for exchanging messages between peers. This example can be enriched with dictionary operations such as `put` a value to a given key, and `get` a value corresponding to a given key (note that `get` corresponds to a *lookup* operation). The key can be any integer ranging from 0 to `maxNetSize-1`, i.e., the size of the virtual search space considered. The value can be any Oz value.

To form the distributed dictionary, we first add a local dictionary `LclDict` to each peer in the network. We also keep a dictionary `TmpDict` to be used to achieve synchronization with the `GetVal` procedure.

```
LclDict = {Dictionary.new}
TmpDict = {Dictionary.new}
```

The following two procedures implement the insertion and respectively the retrieval of a value to/from a distributed dictionary. The `remove` operation can be added similarly.

```
proc {PutVal +Key +Val}
  {OP2PS send(dst:Key msg:put(Key Val) toResp:true)}
end

proc {GetVal +Key ?Val}
  {OP2PS send(dst:Key msg:get(Key) toResp:true)}
  {Dictionary.put TmpDict Key Val}
  {Wait Val}
end
```

Inserting value `Val` at key `Key` into the distributed dictionary translates to storing `Val` at the node responsible of `Key`. Thus, the procedure `PutVal` simply sends the message `put` to the node responsible of `Key`. The `put` message contains the key and the value fields. Procedure `GetVal` sends the message `get` to the node responsible of `Key`. Note that `Val` is temporary stored in `TmpDict` and bound when we receive the answer to the message `get` from the node responsible of `Key`.

Now, in order to be able to interpret these messages, we have to change a bit the loop that processes the incoming messages as follows.

```
% get the message stream and process each received message
for M in {OP2PS getMsgStrm($)} do
  case M of rcvMsg(msg:Payload src:Src dst:_) then
    % store the (key value) pair to the local dictionary
    case Payload of put(Key Val) then
      {Dictionary.put LclDict Key Val}

    % return the value corresponding to Key
    [] get(Key) then Val in
      {Dictionary.get LclDict Key Val}
      {OP2PS send(dst:Src msg:ret(Key Val) toResp:true)}

    % receive the value at key Key
    [] ret(Key Val) then
      {Dictionary.get TmpDict Key Val}
      {Dictionary.remove TmpDict Key}
    else skip end
  else skip end
end
```

If we receive the message `put(Key Val)`, the value `Val` is stored at key `Key` in the local dictionary. If the message is `get(Key)`, the value corresponding to `Key` is returned with the message `ret(Key Val)`. If the message is `ret(Key Val)`, bind `Val` to the variable stored at key `Key` in `TmpDict`.

Note that for simplicity, we assumed that the network does not change, i.e., there is no node joining or leaving the network. A more complete example would include some kind of redundancy and replication.

## 8 Applications using P2PS

This section makes an enumeration of two applications that have been developing using P2PS as underlying network organization. They are the collaboration fruit between UCL, on one hand, and CETIC ([www.cetic.be](http://www.cetic.be)) and respectively INRIA, on the other hand.

### 8.1 Community Panel

Software development is rarely a solo task. The development process of a software, starting from the conceptual design to the code implementation, is the concern of a team involving a lot of people not necessarily located at the same place. Despite of its benefits, collaboration is time consuming. Indeed, some studies reveal that the efforts dedicated to collaboration among developers leave less than half of the workday to do any real coding. Collaborative tools can help to increase the part of the day to do any real coding while still supporting a high level of collaboration. Since from the individual developer's perspective, the IDE (integrated development environment) is where coding take place, why not including collaborative code edition capabilities alongside the editor, compiler and debugger.

The Community-Panel, coming with the peer-to-peer facilities provided by the P2PS library, is a first step toward a collaborative IDE. The objective of this application is to aggregate Oz-developers

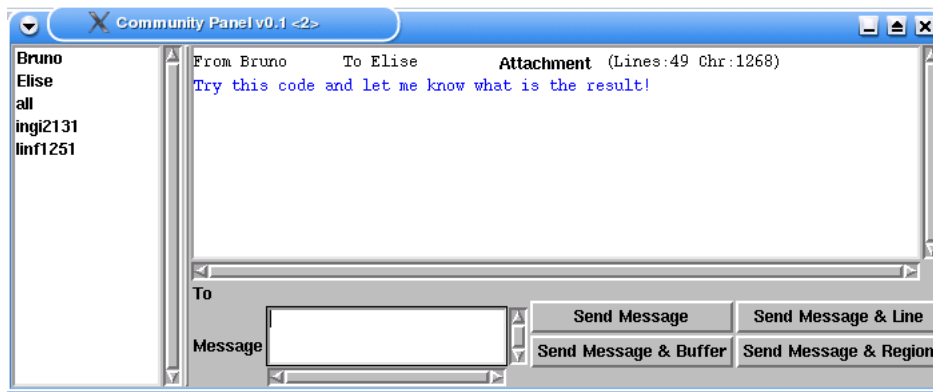


Figure 3: The Community-Panel GUI.

concerned with a common problem in one community and providing the community with tools for real-time collaborative edition.

The targeted functionalities of the Community-Panel are threefold. First, the application provides users with community membership information. This information can be partial or complete, regarding the size of the community and scalability issues, but can be extended on the user's request. Second, the application facilitates the communication between developers by supporting chat-like and instant messaging facilities coming with friends management tools. This allows to meet appropriated community or person according to the user matter, by involving social connections via the friends management tools. Finally, the Community-Panel provides a developing framework for exchanging code in text/binary format but also language entity. For instance, one can imagine to develop an application by adopting a component based architecture where the Community-Panel plays the role of real-time component connector.

Up to now, the Community-Panel demonstrator does not meet all the targeted objectives. One can see that the Community-Panel Interface is composed of 3 areas : the membership, the received messages and the submit areas. The membership area displays all the available groups and the connected users. The received messages area displays all the messages received during the session. The submit area is composed of a text box allowing to write a message and to attach some Oz-code. Once the user received a message with an attachment, she can retrieve the corresponding Oz-Code by clicking on « Attachment » in the received messages area. The retrieved Oz-Code will be inserted in the current buffer of the OPI (Oz Programming Interface) just after the cursor position.

The friends management tools and the language entity sharing are not yet supported but this does not prevent the usage of the Community-Panel. We have developed the core functionalities allowing a first experimentation on collaborative IDE.

## 8.2 PostIt

PostIt is a graphical application where users share a virtual board for publishing *postits*, short time live text, see Figure 4. A postit message is associated with an absolute expiration time. There is only one logical board shared between users which is replicated to every of them, and they are organized as a P2P overlay network. In order to add new *postit* to the logical board, the PostIt application rely on the broadcast primitive provided by the P2PS library. The garbage collection of expired *postits* is performed at each peer by removing the expired postits from its local board thus, eventually, the

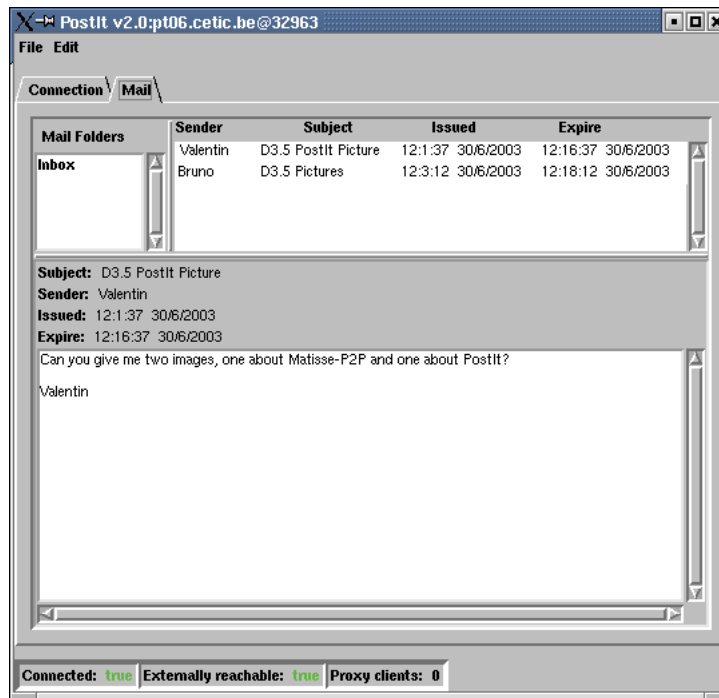


Figure 4: The PostIt GUI.

expired postits will be removed from the whole system.

The demonstrator is currently under extension at INRIA. The purpose of this evolution is to add new functionalities regarding group management and thus, allowing PostIt users to create, join and leave PostIt groups and to communicate within these groups.

## 9 Conclusion

This deliverable is the second report on the progress in WP3 with respect to the development of the peer-to-peer Oz platform. Throughout the document we described our platform, called P2PS – Peer-to-Peer System, focusing on its functionality as well as on its architecture. The P2PS library is developed in Oz-Mozart and it implements Tango, a distributed hash table based algorithm developed in WP2. From its functionality, one can see that P2PS is simple and very practical to construct and work with large scale distributed applications. The feedback – since one year now, from the first release – we are getting from different developers using P2PS allow us to continuously improve its API and functionality.

The P2PS library is available to be (and it is already) used for developing P2P applications. Community-Panel and PostIt are two examples of two realistic applications that use P2PS. More encouraging, P2PS will be employed as a distributed communication environment in research projects to start at UCL and CETIC beginning next year. Although quite in advanced state, P2PS system is not mature yet. As described in the next section, we have undergoing research and further work planned to enrich its functionality, making it even more attractive for developers.

## 10 Further work

This section enumerates the ways we see P2PS evolve from now on. The work on P2PS will continue within projects that we're going to be involved such as EVERGROW ([www.evergrow.org](http://www.evergrow.org)).

**Event based architecture** Based on our experience on developing peer-to-peer middleware, we have started to investigate an event-based architecture focused on global application and distributed software upgrade. An event-based architecture is composed of two actors: there are the clients who produce and consume events, and the event-mapper who maintains a publication-notification relation between event producers and consumers. The role of the event-mapper is to propagate the events published by event producers to the interested event consumers. For that purpose, the event-mapper offers two functionalities: publication and subscription. Via the subscription functionality, an event consumer communicates its interest by exporting event-filters, i.e. event description based on event structure and/or event content. Via the publication functionality, an event-producer can publish some event instance and thus share some information with event-consumers. Once an event is published, the event-mapper performs a consumers selection based on the filters provided through the subscription process and notifies all event-consumers whose filters accept the published event instance.

Two of the benefits of such an architecture are the addressing and time decoupling between producers and consumers. Indeed, a producer does not address directly a consumer and, producers and consumers related to a common event do not have to be available at the same time. Both benefits lead to an architecture composed of loose coupled components.

Usually, the event-based architecture is used in distributed application in order to connect different components of a global application. Our approach differs by investigating first the usage of such an architecture as module linker in a centralized, one process application (i.e. local event-mapper) and then, based on the results obtained for a centralized application, adapt the architecture to distributed application (i.e. global event-mapper).

Providing the event-based architecture with adequate publication and notification semantic will help us to write global applications supporting run-time code upgrade and run-time modification of application's functionality. We intend to apply this architecture organization to the P2PS library.

**Access Points** The current interface for sending messages between nodes in a P2PS network is rather primitive. All messages received by a node appear on a single stream. *Access Points* extend this basic interface to support an arbitrary number of named message queues on each node (this is a generalization of the address, port number addressing scheme popularized by TCP/IP). Do not confuse these *Access Points* found at the node level with `AccessPoint` found at the overlay network level, acting as a gateway to the overlay network

Using this abstraction we provide P2P Ports, which behave like Oz's built-in ports but messages are sent via the Overlay Network.

**RPCs** Oz supports Remote Procedure Calls via logic variables. Unbound logic variables are embedded in the sent message and the caller blocks on these unbound variables when it needs their value. The receiver of the message returns values to the caller simply by binding the logic variables in the usual way.

When using this mechanism with P2PS, binding the embedded logic variable by the callee will cause a direct connection to be made to the caller's node. This may be undesirable, the caller may be behind a firewall or may not have the resources for the connection.

We have extended the P2PS interface (using the Access Points described in the previous section) to provide an overlay RPC. When the callee binds the embedded logic variable the binding will return to the caller via the overlay network, rather than by a direct communication.

**Reliable group communication.** As already mentioned, we are currently finishing the integration of one-to-one reliable communication primitive. Afterwards, we will work on providing one-to-many reliable communication primitive. The challenge here is to provide reliable communication in a relatively highly dynamic network. For this end, there are different algorithms we can choose. The work [10] developed in WP2 on probabilistic algorithms for multicast services in the context of large groups has got our attention.

## References

- [1] MOGUL archive, June 2004. <http://www.mozart-oz.org/mogul> .
- [2] Mozart Consortium, June 2004. Mozart Programming System Release 1.3.0. <http://www.mozart-oz.org> .
- [3] P2PS: Peer-to-Peer System Library, June 2004. Université catholique de Louvain, and CETIC, Belgium. [http://www.mozart-oz.org/mogul/info/cetic\\_ucl/p2ps.html](http://www.mozart-oz.org/mogul/info/cetic_ucl/p2ps.html) .
- [4] Session, June 2003. <http://www.mozart-oz.org/mogul/info/grolaux/session.html> Université catholique de Louvain, and CETIC, Belgium.
- [5] Sockets, June 2003. <http://www.mozart-oz.org/mogul/info/grolaux/sockets.html> Université catholique de Louvain, and CETIC, Belgium.
- [6] Luc Onana Alima, Sameh El-Ansary, Per Brand, and Seif Haridi. DKS(N, k, f): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications. In *Proc. of the 3rd International Workshop On Global and Peer-To-Peer Computing on Large Scale Distributed Systems – CCGRID2003*, Tokyo, Japan, May 2003.
- [7] Bruno Carton and Valentin Mesaros. Improving the Scalability of Logarithmic-Degree DHT-based Peer-to-Peer Networks. In *Proc. of EUROPAR 2004*, Pisa, Italy, August–September 2004. *To appear*.
- [8] Bruno Carton, Valentin Mesaros, and Peter Van Roy. Improving the Scalability of Logarithmic-Degree DHT-based Peer-to-Peer Networks. Technical Report UCL/INFO-2004-01, Université catholique de Louvain, Belgium, January 2004.
- [9] Sameh El-Ansary, Luc Onana Alima, Per Brand, and Seif Haridi. Efficient Broadcast in Structured P2P Networks. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems – IPTPS'03*, Berkeley, CA, USA, February 2003.
- [10] Patrik Eugster and Rachid Guerraoui. Probabilistic multicast. *IEEE DSN, Internatinal Symposium on Dependable Systems and Networks*, 2002.
- [11] J.H.Saltzer, D.P. Reed, and D.D. Clark. End-to-end arguments in system design. In *ACM Transactions in Computer Systems*, volume 2, pages 277–288, November 1984.
- [12] Ion Stoica, Rober Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM*, August 2001.