



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

Deliverable no: D3.5

Peer-to-peer Oz platform (1)

REPORT VERSION: first

REPORT PREPARATION DATE: 2003.06.30

CLASSIFICATION: Public

DELIVERABLE NO: D3.5 DUE DATE: Month 18 DELIVERY DATE: Month 18

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: D3.5
Peer-to-peer Oz platform (1)
Due date: project month 18
Delivery date: 2003-06-30

Responsible Partner: UCL
Participating Partners: UCL, SICS

30th June 2003

By Valentin Mesaros, Bruno Carton, and Peter Van Roy, with input from Erik Klinskog, Donatien Grolaux, and Maria Ramalho

Contents

1	Introduction	3
2	Relation to other workpackages in PEPITO	3
3	P2PS: Peer-to-Peer System	4
3.1	P2PS introduction	4
3.2	Structured overlay network overview	4
3.3	P2PS architecture	5
3.3.1	Com layer	5
3.3.2	Core layer	5
3.3.3	Services layer	6
3.4	P2PS additional characteristics	6
3.5	P2PS Application Programming Interface	7
3.5.1	P2PS.p2pServices class	7
3.5.2	Token (un)wrapping	9
3.5.3	Working with events	9
3.6	A simple example using the P2PS library	11
4	Two applications using the P2PS library	13
4.1	PostIt	13
4.2	Matisse-P2P	14
5	A point-to-point bidirectional session manager with fault detection	15
5.1	The Session Module	15
5.1.1	Application Programming Interface (a short description)	16
5.2	The SocketConnection module	17
6	The Distributed Subsystem	17
6.1	The Oz-DSS integration	18
6.2	The P2PS relation to DSS	18
7	Attached papers	18

1 Introduction

This report presents the progress in workpackage 3 (WP3) with respect to the development of a “peer-to-peer Oz platform”.

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 WP2, in order to provide these languages with peer-to-peer (P2P) abilities. This deliverable is focused on extending Oz-Mozart [2].

We designed and implemented a first prototype of a peer-to-peer Oz platform, as a library, called: Peer-to-Peer System – P2PS [4]. Since Oz-Mozart already provides a distributed programming environment, developing P2PS at the application level was quite quick (i.e., it took us less than 3 months). P2PS implements DKS [8] and GSChord [11, 12], Distributed Hash Table (DHT) based algorithms (developed in P2) that generalize Chord [14]. Their main functionality is to provide the application with the ability to organize itself in a structured overlay network. Moreover, by making use of different algorithms developed in WP2, the P2PS library offers efficient point-to-point, multicast and broadcast communication primitives, as well as low cost overlay network management.

P2PS is delivered as a first prototype software package [4] 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 application programmers to develop P2P applications. First was already the case for the GSChord algorithm. Second is the case for PostIt [5] and Matisse-P2P [13] applications.

The remainder of the document is organized as follows. We continue with the relation of P2PS to other workpackages. In Section 3 we introduce the P2PS library, describing its architecture and API, and showing an example of using it. In Section 4 we present two collaborative applications (namely, PostIt and Matisse-P2P) using the P2PS library. Then, in Section 5 we describe a point-to-point bidirectional session manager with fault detection Oz-Mozart which is used by P2PS as a transport module, providing it with some useful and practical primitives. Finally, in Section 6 we report on the integration of DSS, developed in WP4, into the Oz programming language and the P2PS relation to DSS.

2 Relation to other workpackages in PEPITO

The work in this deliverable is related to other workpackages 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 [2], 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, it can be used by the DSS to provide P2P functionality. Thus, P2PS can be adapted to work directly inside the Communication Service Component – CSC of DSS [10]. On the other hand, the API proposed by P2PS represents a first attempt towards what will be the peer-to-peer API in Oz language.

WP5 Workpackage 5 is in charge with building two 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 4, it is already) used for developing P2P applications in Oz. On the other hand, by using the P2PS library with real applications, allows us to continuously improve its API.

3 P2PS: Peer-to-Peer System

P2PS is a self-contained peer-to-peer library written in Oz, on Mozart. It implements DKS [8] and GSChord [11, 12], peer-to-peer algorithms based on the concept of distributed hash table. We plan to have the P2PS functionality migrate into the language-independent DSS. Prototyping in Oz and migrating selected functionality into the Mozart system has long been used successfully in the Mozart Consortium.

We continue this section by introducing the P2PS library, followed by a brief structured overlay network overview. Then, we present the P2PS architecture, some additional characteristics, and its API. An example of using the P2PS library is also presented at the end of this section.

3.1 P2PS introduction

The main functionality of P2PS is providing the distributed peer-to-peer applications with the ability to organize themselves in a structured overlay network. Due to its organization, the system can perform efficient key based routing, i.e. a message with a key k is routed throughout the overlay network to the responsible of k . Furthermore, P2PS allows the application to form large scale networks by providing it with management and communication primitives whose cost evolves logarithmically with the system size.

The current services provided by the P2PS library can be summarized as follows: connection primitives such as join and leave, communication primitives such as point-to-point, broadcast, and multicast (based on an idea [9] developed in WP2). Since the value of the pair (key , $value$) is to be stored at the application level, P2PS does not provide a lookup primitive by itself. However, DHT operations can be immediately provided by using the communication primitives offered by the P2PS library (see Section 3.6). Moreover, we have undergoing research to develop more communication primitives, as well as provide membership functionality to P2PS.

3.2 Structured overlay network overview

A structured overlay network such as DKS and GSChord is a graph where the nodes represent instances of participants and the edges are connections between the participants. Each participant instance is assigned a unique identifier, uniformly chosen from a key space set. The key space set is totally ordered such that it is possible to define the concept of *successor*: the successor of a node with identifier n is the node who has the smaller identifier greater than n . Each node has a reference to its successor, thus giving the possibility to reach any other node in the system by following the successor links successively. The successor link guarantees the connectivity and coherence maintenance of the overlay network. Note that the responsible of a key k is the node with the smaller identifier greater than k . Thus, routing a message for a key k consists in forwarding the message until it reaches the node responsible of k . Given this graph configuration, the cost to route a message to its destination evolves linearly with the network size while the routing table size stays constant.

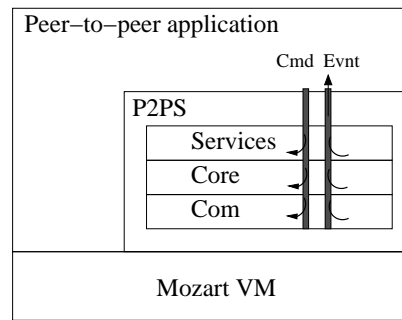


Figure 1: The three-layer architecture of the P2PS library, and its interaction with the application and the transport module (Mozart, here).

Nevertheless, a tradeoff must be made between these two costs (i.e., number of hops and routing table size) in order to obtain a scalable configuration. Thus, by adding well chosen links (called *fingers*) at each node, it is possible to prove (see DKS [8]) that the number of routing table entries and the number of hops requested to deliver a message follow the network size logarithmically.

3.3 P2PS 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.

3.3.1 Com layer

The purpose of the `Com` layer is to provide the upper layer (i.e., `Core`) with point-to-point communication and channel establishment primitives at the level of the underlying network. It also advertises temporal and permanent channel failures. The `Com` layer is based on a transport module providing it with basic communication channels and channel failure detection. For the moment, P2PS uses the distribution module of Mozart as its transport layer, by employing the communication abstraction described in Section 5.

Since a structured peer-to-peer network is composed of active entities, each peer must offer an access point. The description of the access point provided by a peer A can be used by a peer B in order to establish a communication channel between them. We chose to have the `Com` layer implicitly hiding the transport medium it employs for a better separation between the layers.

Besides channel establishment, the `Com` layer provides the `Core` layer with point-to-point message sending and receiving primitives in the underlying network. Moreover, to give the ability to take action when a channel fails, the `Com` layer advertises the `Core` layer about temporary failure (e.g., congestion) and permanent failure of channels (i.e., process crash).

3.3.2 Core layer

The `Core` layer, as its name indicates, is the central component of the P2PS library. It implements DKS [8] and GSChord [11] algorithms. Its purpose is threefold: maintain the routing table and the successor list regardless the nodes joining and leaving, route key based messages to their responsible, and implement node join and leave mechanisms.

The join mechanism consists in, given an entry to the system, finding the right place of the joining node (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 peer-to-peer overlay network. The leave operation is much simpler and consists only in advertising its connected peers about the leave.

The message routing algorithm is based on what is called the *key lookup* primitive of DKS and GSChord. 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.

Another operation is the routing table maintenance. Instead of correcting the routing table explicitly by probing periodically, the routing table is corrected implicitly (as described in [8]) when the peers are actually using the network. 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).

3.3.3 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 in 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 point-to-point message send, and message broadcast and multicast operations. Note that in the current version no provision is available yet to guarantee message delivery (see Section 3.4).

The application can interact with the *Services* layer by invoking the required methods directly or by simply reading and writing information on the three available streams: *message*, *event*, and *command* (see Section 3.5.3).

3.4 P2PS additional characteristics

Support for 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 the *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 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 they 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 `firewall` 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.

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, in P2PS, we use the *correction-on-use* mechanism proposed in [8] 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 messages for an unused network.

Message send reliability The P2PS library provides reliable message sending, except in the situation where nodes on the path between sender and receiver disconnect during the message send. Currently, when a node disconnects from the system while it is about to forward a message, this message can be lost. In the future version of P2PS we will extend the library to achieve complete reliability, even in the face of frequent node failures.

Nevertheless, an application using the present P2PS can implement, if necessary, it's own level and flavor of reliability on top of the library, if necessary.

3.5 P2PS Application Programming Interface

This chapter enumerates and describes the classes and their accessible methods of the P2PS library, thus familiarizing one with its API. For the moment, the P2PS library has a simple structure containing only one class, namely `P2PServices`, and a couple of helpful procedures.

3.5.1 P2PS.p2pServices class

`P2PServices` is the class offering P2P services.

```
init(nid: +NId <= _
     pn: +PN <= _
     ctrlstrm: +CS <= _
     type: +Type <= gschord)
```

Initialize the class `P2PServices` and create a P2P node of type `Type`, where `Type` is the type of overlay network to be used (i.e., `gschord` for `GSchord` and `dks` for `DKS`). It also creates an access point for this node on port number `PN`, if specified; otherwise a random port number is chosen. `NId` is the node identifier (described in Section 3.2). `CS` is the command stream. It is used to (un)register for events (see Section 3.5.3).

```
createAccess(pn: +PN <= _
            tn: ?Tn <= _)
```

Create an access point for this node on port number PN , if specified. Otherwise, choose a random port and bind it to PN . Return the corresponding token T_n of this node (see Section 3.5.2 for more on token). This method is called before a node issues a join, and after it has already called the leave method. Raise exception `p2ps(couldNotCreateAccess pn:)` when there is a problem creating the access. Raise `p2ps(accessAlreadyCreated pn:)` when an access is already created on that port number PN .

```
join(tn: +Tn
     nid: +NId <= _)
```

Join this node to the system via the node at the address indicated by the so called *token* T_n (see Section 3.5.2). Note that a node can join the system via any other node already in the system. Raise exception `p2ps(joinFailed)` when the token has not the proper form or refers to a dead site. NID is the node identifier (described in Section 3.2). If NID is specified and this identifier is already used by another peer then the chosen NID will be the next free one. Raise exception `p2ps(protocol_mismatch name:N version:V arity:A ns:NS)` when the parameters of the joining node do not match those of the system (i.e., the protocol employed and its version, the arity, the network size).

```
leave
```

Detach this node from the system. Close the access point at this node.

```
sendMsg(to: +NId
        msg: +Msg
        tr: +Tr <= false)
```

Send message Msg to node identified by NID (i.e., when Tr is **false**). If Tr is **true**, NID is considered to be a key and Msg is sent to the key's responsible. The send is asynchronous. In the current version no provision is available yet to guarantee its delivery (see Section 3.4). Raise exception `p2ps(nodeNotJoined)` if the node is not part of any system.

```
multicastMsg(to: +LNID
             msg: +Msg
             tr: +Tr <= false)
```

Send message Msg to nodes found in list $LNID$. If Tr is **true**, the *ids* are considered to be keys and Msg is sent to the keys' responsables. The send is asynchronous. In the current version no provision is available yet to guarantee its delivery (see Section 3.4). Raise exception `p2ps(nodeNotJoined)` if the node is not part of any system.

```
broadcastMsg(msg: +Msg)
```

Send message Msg to all nodes in the system. The send is asynchronous. In the current version no provision is available yet to guarantee its delivery (see Section 3.4). Raise exception `p2ps(nodeNotJoined)` if the node is not part of any system.

```
getToken(tn: ?Tn)
```

Return the address of this node as a token T_n (see also Section 3.5.2).

```
getAddr(ip: ?IP
        pn: ?PN)
```

Return the IP address and the port number of this node.

```
getNodeId(nid: ?NId)
```

Return the *id* of this node.

```
getMsgStrm(ms: ?MS)
```

Return a stream on which the received messages will be put. The messages have the form of the record `rcvMsg(fm:F msg:M)`, where *F* denotes the sender node and *M* the message content. Note that all messages received before binding *MS* are not available.

```
getEvtStrm(es: ?ES)
```

Return a stream on which events will be sent. Note that all messages received before binding *ES* are not available (see also Section 3.5.3).

3.5.2 Token (un)wrapping

A token is a placeholder that carries the address information of a node in P2PS. The node address is provided by the underlying communication level and its format can vary (e.g., IP and port number, DSS ticket).

In order to join a P2PS system, a node has to specify the access point of any other node already in the system in the form of a token. One can either obtain the token from the interested node, or he can build it up with `P2PS.addressToToken` procedure, given the node's IP address and port number. The opposite procedure is `P2PS.tokenToAddress`.

```
addressToToken(ip: +IP
               pn: +PN
               tn: ?Tn)
```

Given an address as the IP and the port number PN, return the corresponding token Tn.

```
tokenToAddress(tn: +Tn
               ip: ?IP
               pn: ?PN)
```

Given a valid address based token Tn, return the address as the IP and the port number PN. Otherwise, raise exception `p2ps(notAddressBasedToken tn:)` when the token is not valid.

3.5.3 Working with events

Events represent asynchronous actions that occur at different levels at a node (i.e., *serv*, *core*, and *com*). An application can watch the events and *react* accordingly.

(Un)Register an event In order to have access to events, an application has to register for those it is interested in via the command stream CS provided at method `init` of class `P2PS.p2pServices`. To register for an event one has to send the following record on the command stream: `c(lv:Level cmd:Command)`, specifying the corresponding level (i.e., `serv`, `core`, and `com`) and the command `regEvt(evntLst: +EvtLst)`, where `EvtLst` is a list of events to register to.

To unregister an event one has to send the following record on the command stream: `c(lv:Level cmd:Command)`, specifying the corresponding level (i.e., `serv`, `core`, or `com`) and the command `unregEvt(evntLst: +EvtLst)`, where `EvtLst` is a list of events to unregister.

Events supported Currently only the `core` level supports events; they are enumerated as follows.

`alone`

Issued when the node leaves the system. This event is registered by default.

`connected`

Issued when the node becomes connected to a P2PS system. It has the form `connected(firewall: ?B)`, where `B` is a boolean indicating whether the node can be (**false**) or not (**true**) accessed directly by other nodes in the system. In the latter case, this node uses its successor as a proxy in order to be part of the system (see Section 3.4 for more on proxy node). This event is registered by default.

`newcon`

Issued when this node is used by another node identified by `Nid` as a proxy to enter the system (see Section 3.4 for more on proxy node). It has the form `newcon(Nid)`.

`newft`

Issued when the finger table of this node has changed. It has the form `newft(nid:?Nid table:?T)`, where `T` is a record of node *ids* representing the fingers of this node.

`newpred`

Issued when this node has a new predecessor node. It has the form `newpred(nid:?Nid)`.

`newsuc`

Issued when this node has a new successor node. It has the form `newsuc(nid:?Nid)`.

`proxycon`

Issued when a node connects/disconnects to/from this node. It has the form `proxycon(?NP)`, where `NP` is the number of nodes using this node as proxy (see Section 3.4 for more on proxy node).

3.6 A simple example using the P2PS library

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`. In this example `node3` sends a multicast message to `node1` and `node2`, and a point-to-point message to `node1`. Moreover, we will extend this example with DHT operations. The code runs directly in the Oz Programming Interface.

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. It runs a loop over the message stream and displays the messages addressed to it.

```
declare /* the node1 */
[P2PS]={Module.link ["P2PS.ozf"]}

MyIP  % IP address
MyPN  % port number

% create a node with id=1 on port 3001
CP2PS = {New P2PS.p2pServices init(nid:1 pn:3001)}

% get the Ip and port# of this node
{CP2PS getAddr(ip:MyIP pn:MyPN)}

{System.showInfo `node at address ip:~#MyIP#` and pn:~#MyPN}

% get the message stream and show each received message
thread
  for M in {CP2PS getMsgStrm(ms:$)} do
    {Show M}
  end
end
```

We create `node2` with `id` 16. It joins the system via `node1` specifying its address and port number. Further on, we run a loop to wait and show the messages sent to this node.

```
declare /* the node2 */
[P2PS]={Module.link ["P2PS.ozf"]}

% create a node with id=16 on any port number
CP2PS = {New P2PS.p2pServices init(nid:16)}

% wrap address into token
Tn = {P2PS.addressToToken `127.0.0.1` 3001}

% connect to the system via a node (node1) by providing its Ip and port#
{CP2PS join(tn:Tn)}

% get the message stream and show each received message
thread
  for M in {CP2PS getMsgStrm(ms:$)} do
    {Show M}
  end
end
```

We create `node3` with `id` 24. `node3` chooses to join the system via `node1` (note that it could have done the join via `node2`) specifying its address and port number. Further on, it sends a multicast message to `node1` and `node2`, and a point-to-point message to `node1`.

```
declare /* the node3 */
[P2PS]={Module.link ["P2PS.ozf"]}

% create a node with id=16 on any port number
CP2PS = {New P2PS.p2pServices init(nid:24)}

% connect to the system via a node (node1) by providing its Ip and port#
{CP2PS join(tn:{P2PS.addressToToken `127.0.0.1` 3001})}

% send one message to node 1 and node 16
{CP2PS multicastMsg(to:[1 16] msg:hello)}

% send one message to node 1
{CP2PS sendMsg(to:1 msg:sasa)}
```

The three-peer system example shows how to build up a P2P system and how the P2PS library can be used for message exchange between peers. Moreover, this example can be enriched with DHT operations such as *put* a value to a given key, and *get* a value corresponding to a given key (note that the *get* corresponds to the *lookup* primitive). The key can be any integer ranging from 0 to $N - 1$, where N is the size of the virtual search space. The value can be any Oz language variable. 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}
  {CP2PS sendMsg(to:Key msg:put(Key Val) tr:true)}
end

proc {GetVal Key Val}
  {CP2PS sendMsg(to:Key msg:get(Key Val) tr:true)}
end
```

Inserting the value `Val` at the key `Key` into the distributed dictionary translates to storing `Val` at the node responsible for `Key`. Thus, the procedure `PutVal` simply sends the message `put` to the node responsible for `Key`. The `put` message contains the key and the value fields. Similarly, the procedure `GetVal` simply sends the message `get` to the node responsible for `Key`. The value in the `get` message will be bound by the node responsible for `Key`. The binding is done transparently by the distribution of Mozart.

To form the distributed dictionary, we add a local dictionary `LclDict = {Dictionary.new}` to each node in the P2P system. Now we will see how the `put` and `get` messages are interpreted at a peer node. For this, we modify the loop that processes the incoming messages. Thus, if the received message is `put(Key Val)` then the value `Val` is stored at key `Key` into the local dictionary. On the other hand, if the received message is `get(Key Val)` then `Val` will be bound to the value corresponding to the key `Key` in the local dictionary.

```
thread
  for M in {CP2PS getMsgStrm(ms:$)} do
    case M of rcvMsg(msg:C fm:_) then
      case C of put(Key Val) then
        {Dictionary.put LclDict Key Val}
```

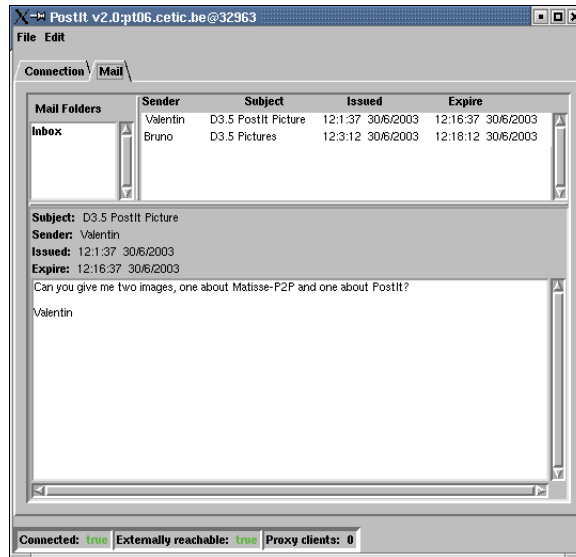


Figure 2: Screenshot of the PostIt application.

```

[] get(Key Val) then
  {Dictionary.get LclDict Key Val}
  else skip end
else skip end
end
end
end

```

When extending the three-peer system example with DHT operations, we assumed, for simplicity, that the system does not change (i.e., there is no node coming or leaving). A more complete example would include key/value pair redistribution when nodes are joining and leaving.

4 Two applications using the P2PS library

4.1 PostIt

PostIt [5] is a graphical application where users share a virtual board for publishing *postits* – short time live texts – see Figure 2. 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 avoid single points of failure, and to achieve scalability there is not a single responsible of the board. Adding a postit to the logical board is done monotonically. When a peer issues a new postit it adds it to its local board (i.e., its view). Then, it advertises this operation together with the new postit to all the other peers. This announcement is done by using the broadcast primitive of P2PS. However, using the P2PS broadcast primitive is not enough to ensure that all peers will eventually have the same view of the shared board. In order to converge towards a common view of the shared board, PostIt synchronizes the views each time the network changes. That is, each time a peer has a new successor, they synchronize their views with each other. After the synchronization procedure, the missing postits at one side are broadcasted to all the other peers.



Figure 3: Screenshot of the Matisse-P2P application.

Removing a postit from the shared board is much simpler. Each peer removes the expired postits from its local board thus, eventually, the expired postits will be removed from the whole system.

The PostIt application is implemented in Oz-Mozart using the P2PS library. It has already been successfully tested in LAN and MAN, even with peers laying behind firewall.

4.2 Matisse-P2P

Matisse-P2P [13] is based on a stand-alone application, called Matisse [1]. The original Matisse is a drawing tool that intends to promote the drawing creativity of young children and enable them to feel confident with their own creations. In Matisse the user can choose between drawing patches or lines. Drawing lines requires a greater mastery and technique than drawing color patches. This is why a novice user will rather draw color patches than fine lines. Matisse-P2P uses the same drawing idea (see Figure 3). In addition a means has been devised to enable multiple users to draw synchronously in a virtual drawing space making it a nice peer-to-peer collaborative application. Matisse-P2P is currently being developed in Oz and it employs the P2P primitives of the P2PS library.

Matisse-P2P can be used as a demonstrator application, showing different capabilities provided by the third generation P2P platform developed in PEPITO project. The following features of the P2P platform can be highlighted:

Group communication primitives Each of the shared drawings in Matisse-P2P is associated with a group and functionality is provided to add new groups, remove existing groups and to participate (subscribe) in an on-going collaborative drawing session.

Robustness to network failures This is in order to make sure that users feel confident in using the application and network failures will actually take place without users taking notice of them.

Minimum system maintenance The fact that the application is implemented on a decentralized (not server based) platform, will allow it to be deployed with minimum overlay costs.

Low infrastructure cost for the end user Matisse-P2P is an application that can run either on a desktop PC or on an hand-held computing device. The peer-to-peer platform will enable to shift computing efforts through the existing nodes of the platform – if need be – in order to guarantee anywhere, anytime access to the Matisse-P2P application.

5 A point-to-point bidirectional session manager with fault detection

The P2PS library allows nodes to freely join and leave the overlay network. However, as communication in the underlying network is based on point-to-point channels, nodes have to connect and disconnect successively from other nodes; i.e., they require a session communication manager. Since, the built-in data types of Oz don't provide a direct way to implement sessions, a Session module has been created specifically for this very purpose.

The Session module attaches a state stream to each running communication session. The state stream reflects the current knowledge of the *health* of the network communication, providing the information required by applications to tolerate network faults. Thus, the use of the Session module made the detection of network faults in P2PS much easier. Moreover, the Session module provides P2PS with two additional useful features. That is, the possibility to close a connection (necessary in a self-organized network), and the fact that a connection is bidirectional (very useful to exploit the symmetry in GSChord). The Session module is written in Oz, on Mozart, and is available as a library [6].

Although Oz provides natively a connection mechanism based on the exchange of an encoded string, this string is usually brought from one Oz process to another using an external medium such as a web server. In the case of P2PS, one can not rely on such external medium. Thus, the SocketConnection (see Section 5.2) module has been developed to use a standard combination of IP address and socket number to make Oz processes connect together. Consequently, the P2PS nodes only need to know an IP and a socket number to connect together; there is no more external medium required. The SocketConnection module is written in Oz, on Mozart, and is available as a library [7].

5.1 The Session Module

The Session module provides a communication channel respecting the following set of properties. For a complete description of this module refer to the Session package in the attached papers.

- ▷ A Session instance obtained by `Session.new` represents one side of a communication.
- ▷ When two instances are connected together, they offer a bidirectional communication channel between these two points. Messages are received on a stream. There is only one reception stream per session instance that receives all messages from all successive connections of this instance.
- ▷ At maximum two Session instances can be connected together at any time.
- ▷ Any of the two sides of a session communication channel can break the connection. Each side can reconnect back or to other session instances.
- ▷ Each session instance also provides a *free-for-all communication channel* where anyone can send messages freely without having to connect. These messages are received on a different stream. An example is an application which wants to provide a remote control to several remote clients, but only one at a time. With the Session module, the application can use a single session instance to let only one remote client connect at a time. When the client wants to stop working with the application, it can close the session, and another client can then take over. If a client is already connected to the application and another one wants to connect to it also, it can use the *free-for-all channel* to notify the application about that. If there are several applications the client wants to connect to successively, it can do so using a single session instance.

- ▷ Each session instance produces a link state stream. Connections change of the communication link with the currently connected session and disconnections generate messages on that stream. This can be used to create fault-tolerant applications without using the Mozart Fault module. The possible messages on that stream are:

`connect` This session instance has just been connected to another session instance. Messages can be sent to the other site.

`disconnect` This session instance has been disconnected by the other session instance. No message can be sent while disconnected.

`tempFail` There is a communication problem that temporarily prevents the messages from arriving to the other site. A `tempFail` may change into a `permFail` or into `ok`, or stays forever (until the site decides to break the session). Note that the application can still send messages in this state. They will be buffered and sent correctly when the communication turns `ok` or dropped if the communication is broken.

`permFail` The other site was detected crashed. It is the same as `disconnect` except that there is no use to try and contact the other site anymore; it is dead forever.

`ok` After a `tempFail`, this state means the communication is fine again, and messages are normally arriving at the other site.

5.1.1 Application Programming Interface (a short description)

```
{Session.new ?S}
```

Return a new session instance.

```
{Session.connect S1 S2}
```

Connect `S1` with `S2`. Raise an exception if `S1` or `S2` are already connected to another session. Raise an exception if `S2` is unreachable.

```
{Session.disconnect S}
```

Disconnect `S` from whatever other session it was connected to. This is a synchronous disconnection; i.e., it waits for all pending messages to arrive before disconnecting.

```
{Session.break S}
```

Disconnect `S` from whatever other session it was connected to. This is an as soon as possible disconnection, and some messages may be dropped in the process.

```
{Session.getStream S ?Xs}
```

Return the stream of messages sent by the sessions connected to `S`.

```
{Session.getStateStream S ?Xs}
```

Return the stream of communication states of S with the other sessions it is successively connected to. This stream can be composed of one of the following atoms: `connect`, `disconnect`, `tempFail`, `ok`, `permFail`.

```
{Session.getState S ?T}
```

Return the current state of S .

```
{Session.sSend S M}
```

Send M to the reception stream of the session S is connected to in a synchronous way: this command blocks until this message has arrived on the reception stream of the other site. Raise an exception if S is not currently connected.

```
{Session.aSend S M}
```

Send M to the reception stream of the session S is connected to. Doesn't block. Raise an exception if S is not currently connected.

```
{Session.isConnected S ?B}
```

Return `true` if S is currently connected, `false` otherwise.

5.2 The SocketConnection module

The `SocketConnection` module provides a socket-based mechanism to establish connections between independent Oz processes. One process (called server) opens a socket with which other sites (called clients) can connect to establish a connection. For a complete description of this module refer to the `Sockets` package in the attached papers. Two different types of socket connections are supported:

One-to-one connections Allow to establish a single connection only.

Many-to-one connections Allow multiple connections with the same socket to the same value. Values for many-to-one connections are offered through gates.

6 The Distributed Subsystem

The Distribution Subsystem – DSS [10] 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 behavior.

During the first half of the second year, focus has served from basic services towards P2P services. Out of the three properties that define a P2P node, simultaneously being a server, a client, and a router, the DSS could act as a server and a client. Naturally, the platform had to be extended with the capability of routing. This section will briefly describe the integration and Oz-DSS and how the P2PS library is related to it.

6.1 The Oz-DSS integration

Work has started on the integration of Mozart and the DSS. In a prototype version [3] 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 Mozart of today. However, it 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 the next autumn, the latest available Mozart system will be enhanced with the distribution support of the DSS, thus making it a fully-fledged P2P programming platform.

6.2 The P2PS relation to DSS

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.

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. Second, the P2PS module can use the advanced point-to-point communication abstractions provided by the DSS. Thus avoiding redundant implementation efforts.

7 Attached papers

1. Maria F. Ramalho. What would children unable to attend school learn from constructing OZ TALES. In *Proc. of EUROLOGO2003*, August 2003 (*to appear*).

The following package is available at the indicated URL:

2. P2PS03, June 2003. <http://wikis.info.ucl.ac.be:8080/index.php/Peer2PeerSystem>
Universtité catholique de Louvain, and CETIC, Belgium.

References

- [1] Matisse, January 2003. <http://studiolab.io.tudelft.nl/tinytools/Matisse> Technische Universiteit Delft, Netherlands.

- [2] Mozart Consortium, June 2003. Mozart Programming System Release 1.2.5. <http://www.mozart-oz.org>.
- [3] Oz-DSS, June 2003. <http://dss.sics.se> Swedish Institute of Computer Science, Sweden.
- [4] P2PS, June 2003. <http://wikis.info.ucl.ac.be:8080/index.php/Peer2PeerSystem> Université catholique de Louvain, and CETIC, Belgium.
- [5] PostIt, April 2003. <http://www.info.ucl.ac.be/~valentin/postit/ReadMe.html> Université catholique de Louvain, and CETIC, Belgium.
- [6] Session, June 2003. <http://www.mozart-oz.org/mogul/info/grolaux/session.html> Université catholique de Louvain, and CETIC, Belgium.
- [7] Sockets, June 2003. <http://www.mozart-oz.org/mogul/info/grolaux/sockets.html> Université catholique de Louvain, and CETIC, Belgium.
- [8] 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.
- [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] Erik Klintskog, Zacharias El Banna, and Per Brand. A generic middleware for intra-language transparent distribution. Technical Report T2003:01, Swedish Institute of Computer Science, Kista, Sweden, January 2003.
- [11] Valentin A. Mesaros, Bruno Carton, and Peter Van Roy. Improving and Generalizing Chord. presented at *Global Computing Workshop*, Rovereto, Italy, January 2003. Available at <http://www.info.ucl.ac.be/~valentin/papers.html>.
- [12] Valentin A. Mesaros, Bruno Carton, and Peter Van Roy. S-Chord: Using symmetry to improve lookup efficiency in Chord. In *Proc. of the 2003 International Conference on Paralell and Distributed Processing Techniques and Applications – PDPTA'03*, Las Vegas, USA, June 2003.
- [13] Maria F. Ramalho. What would children unable to attend school learn from constructing OZ TALES. In *Proc. of EUROLOGO2003*, Porto, Portugal, August 2003 (*to appear*).
- [14] Ion Stoica, Rovert Morris, David Karger, Frans M. Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. Technical Report TR-819, MIT, March 2001.