P2PS: Peer-to-Peer Development Platform for Mozart*

Valentin Mesaros[1], Bruno Carton[2], and Peter Van Roy[1]

[1] CS Department, Université catholique de Louvain, Louvain-la-Neuve, Belgium
{valentin, pvr}@info.ucl.ac.be
[2] CETIC, Charleroi, Belgium bruno.carton@cetic.be

Summary. Recently, development of peer-to-peer (P2P) applications has been giving a paramount attention mostly due to their attractive features such as decentralization and self-organization. Providing the programmer with the "right" platform for developing such applications became a challenge. In this paper we describe the functionality of `P2PS`, a platform for developing P2P applications in Mozart. The `P2PS` platform provides the developer with a means for building and working with P2P overlay applications, offering different primitives and services such as group communication, efficient data location, and dealing with highly dynamic networks. `P2PS` implements Tango, an efficient algorithm for constructing structured P2P systems. It is delivered as a library and already made public, being used as underlying structure for different P2P applications.

1 Introduction

With the advent of popular peer-to-peer (P2P) applications and systems such as Gnutella (gnutella.wego.com) and Napster (www.napster.com), the development of P2P systems has become important and even a research topic. The main reasons for this "rush" is due to the practical and useful features and objectives of P2P computing, e.g., scalability, self-organization, decentralization. The very idea behind the peer-to-peer concept is the fact that the processes participating in a distributed computing can exchange information directly, without passing through a central point. Thus, they become *peers*. A peer can be client, server, and router at the same time. Generally speaking, peers have equal capabilities and eventually equal responsibility.

In this paper we present some of our ongoing work within the framework of extending Mozart/Oz (www.mozart-oz.org) to reflect new programming abstractions that use different distributed algorithms in order to offer P2P abilities. We describe the functionality of the `P2PS` [1] peer-to-peer development platform. The `P2PS` platform provides the developer with the ability of building and working with P2P over-

* This work was funded at UCL by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under IST-2001-33234 PEPITO, and at CETIC by the Walloon Region (DGTRE) and the E.U. (ERDF and ESF).

lay applications, offering him different P2P primitives and services such as group communication, efficient data lookup, and fault-resilience. Although independent of the underlying P2P technology, `P2PS` currently only implements Tango [2]. Tango is a peer-to-peer algorithm that we developed to better structure relative exponential networks to increase their scalability. It extends and improves Chord [3], and thus it can be included into the category of structured P2P systems (see Section 2).

Mozart already provides the programmer with an advanced interface for developing distributed applications. However, the underlying distribution layer of Mozart is based on a client-server model which may lead to scalability problems with respect to the number of processes involved in the computation. There is ongoing work [4, 5] to reduce some of these problems. On the other hand, the `P2PS` development platform inherits all its functionality from the P2P algorithms. For example, it can provide full connectivity (though, multi-hop) between all nodes within the network, and this with a logarithmic number of physical connections per node. Moreover, the loosely coupled model together with the management of highly dynamic networks make of `P2PS` the right choice to develop P2P applications in Mozart.

Related to our work, there exists several research projects for P2P development platform. One is the Chord project (www.pdos.lcs.mit.edu/chord) written in C++ and based on the Chord algorithm. Another platform is FreePastry (freepastry.rice.edu) written in Java and based on the Pastry [7] algorithm. As `P2PS`, both platforms are based on a structured P2P system and aim to build scalable, robust distributed systems. They both offer a programming interface based on the "common API" [6]. `P2PS` provides rather a combination between layers tire 0 (i.e., key-based routing) and tire 1 (e.g., multicast and broadcast) of the "common API". `P2PS` as well as Chord project and FreePastry are in ongoing research and they all three are more or less similar with respect to the services they offer. One thing that differentiate them from one another is the programming language they are written in. Hence, we believe that, given the expressiveness of Mozart/Oz, `P2PS` represents an attractive choice for writing P2P applications with.

JXTA [8] and JXTAnthill [9] are two other development platforms for P2P applications. JXTA defines a set of basic protocols for a number of P2P services such as discovery, communication, and peer monitoring. JXTAnthill is written on top of JXTA and it implements algorithms rooted in complex adaptive systems, based on the behavior of ants. The difference between `P2PS` and these two platforms is mostly based on the P2P algorithms they each implement. While `P2PS` is based on algorithms offering strong data lookup guarantees, this is not the case for JXTA and JXTAnthill.

The remainder of this document is organized as follows. We continue by briefly recalling the principles of structured P2P systems. In Section 3 we present the main functionality provided by `P2PS`. In Section 4 we describe the internal architecture of `P2PS`. In Section 5 we show how to write a simple application for `P2PS`, and in Section 6 we describe a more realistic application that uses `P2PS`, and then conclude.

2 Structured P2P Systems

In this section we briefly recall the principles and notations of the structured P2P networks. Unlike unstructured P2P systems like Gnutella, whose overlay topology is ad-hoc, structured P2P systems organize their overlay by following well specified rules in order to improve overall efficiency. A key challenge in building P2P systems is providing means for efficient location of information distributed across a large number of processes (or nodes) of a highly dynamic network. We take the Chord algorithm as a case study since Chord is one of the first P2P algorithms based on the idea of Distributed Hash Table – DHT, and also because Chord and Tango have many commonalities.

There are three main characteristics of a P2P structured system. First is the fact 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' *id*s. (Hereinafter, we will use the term *node* to refer both to the node itself and to 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. Second 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, i.e., the key *lookup*, is provided with clear guarantees. For instance, while in Gnutella a flooding-based algorithm is used, leading to network resource waste, in Chord and Tango the lookup for a key will not take more than a certain maximum number of hops and messages, i.e., $O(logN)$, where N is the maximum number of nodes in the overlay. Third is the system's resilience to node failures and its ability to self-organize face to the network's dynamics. That is, when nodes join or leave the network, the nodes pointing to them will adapt their local routing tables in order to guarantee overall efficient lookup. Furthermore, since the system is totally decentralized, there is no risk for single points of failure to occur.

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*) to some other nodes inside a *routing table*. Given an identifier space of size N, beside the references to its predecessor and successor, each node in the Chord system stores $logN$ 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.

3 Functionality

In this section we present the main functionality provided by our P2P platform, called `P2PS`: *Peer-to-Peer System*. The main functionality of `P2PS` is offered 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. Although implementing the Tango algorithm, `P2PS` offers an API that can apply to any structured P2P system. Thus, the programmer does not have to worry about the underlying details. For more info on the API, the reader should refer to the `P2PS` tutorial [1].

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 intend to provide basic P2P primitives on top of which more specialized services will be built. Dictionary functionality such as 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. Furthermore, we have undergoing research to extend the functionality of `P2PS`.

3.1 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 the fact that an `AccessPoint` is created for this node. (For the description of the access point, see Section 4.1.) 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 number of nodes in the overlay, 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 other nodes to connect and join the overlay network. Furthermore, the node is provided with message and event input streams on which messages from other nodes and respectively different node and network events will eventually be accessible.

3.2 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 node within the system does not depend on the entry point it used to get into the network. The system will self-organize in order to guarantee overall efficiency (see Section 4.2 for details). 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 input streams.

3.3 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 does a gracefully leave. Thus, underneath, a node will run a simple protocol to disconnect it 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.

3.4 Message Sending and Receiving

`P2PS` provides end-to-end communication primitives. That is, sending messages from one peer node to another throughout the overlay network. Due to its organization, the system performs efficient key based routing. Thus, 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. In `P2PS` the message sending and receiving are asynchronous. Nevertheless, the reliable send can be made synchronous. In the following we describe how to send messages by using different communication primitives. In all cases, receiving a message at a node implies reading the message input stream at that node. The messages addressed to a node will appear on its associated message stream.

One-to-one communication. This primitive is to be used to send messages from one node to another one, throughout the overlay network. It is important to note that in `P2PS` one can choose 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 will eventually always reach its destination (since there will always be a node responsible for any key), in the latter the destination may simply not be present. Both flavors of message delivery are useful in practice. On the other hand, one can choose between a best-effort send or a reliable send. In the case of a best-effort send, although generally the message will reach its destination, there are situations when the message may be lost, e.g., due to the overlay network dynamics. In the case of a reliable send the message will be delivered to the destination; otherwise, its loss will be signaled to the sender. To send one-to-one messages in `P2PS`, one will use the method `send` for best-effort send and the method `rsend` for reliable send.

One-to-many communication. Another communication primitive that `P2PS` provides is one-to-many, where simple and efficient `broadcast` and `multicast` is provided. Both protocols employed are based on an idea [10] that exploits the tree structure of a Chord-like system. 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, i.e., explicit multicast. As in the case of one-to-one primitive, in the case of multicast one can choose to send the message either to nodes' responsibles, or directly to the nodes whose *id*s equal those in the destination list.

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

3.5 Monitoring

In a dynamic network, as in P2P overlay networks, being aware of the status and 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. Hence, in `P2PS` we decided to provide a set of events on the event input stream associated with the peer node. These events indicate changes on the connections with the node's neighbors. This way, for example, if the successor of a node has changed, the application may do replication on the new successor.

Another way of monitoring a 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 node. For example, one can obtain 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 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 sending/receiving and channel establishment operations.

4.1 COM Layer

The COM layer is in charge with interfacing with the underlying physical network. Basically, COM provides the Core layer with communication functionality through a common API, regardless the underlying transport protocol employed. The functionality provided by the COM layer is: access point creation, connection establishment, basic communication primitives, and fault detection.

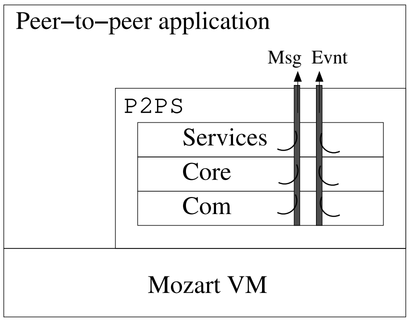

Fig. 1. The three-layer architecture of a P2PS node, and its interaction with the application and the transport module (here provided by the distributed layer of Mozart).

Access point creation. We define an access point to be an addressable entry point of a node. It is the COM layer who defines the form and the meaning of an access point. Moreover, the representation of an `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 addressable entry point for a peer node. Then, a peer node can publish its access point, allowing remote connections to it, and thus providing an access point to the overlay network itself.

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

Basic communication primitives. The basic communication primitives provided are `send` message and `receive` message. They are point-to-point primitives providing reliable transfer over connections established via an `AccessPoint`.

Fault detection. The fault detection primitives provide a means for detecting two types of network anomalies relative to point-to-point connection, i.e., permanent faults and temporary faults. Given the high dynamics of a P2P network, this functionality is very important. For this end, P2PS uses indirectly the distribution functionality of Mozart.

4.2 Core Layer

An overlay network topology can be viewed as a graph composed of arcs and nodes. The Core layer provides high-level connectivity primitives between 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 [2] algorithm. Its purpose is threefold: implement node join and leave mechanisms, route key based messages to their responsibles, and maintain the routing table and the successor list regardless the nodes joining and leaving, thus guaranteeing overlay efficiency.

Joining/leaving a network. Given an entry point to the system (i.e., `AccessPoint`), the join mechanism consists in finding the right place for the joining node within the overlay network (i.e., between its successor and predecessor), and establishing a communication channel with its neighbors. Obviously, the predecessor and successor of the joining node will be affected by the join operation and therefor 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 after being itself correctly inserted into the overlay. 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 the key lookup primitive of the P2P algorithm employed (i.e., Tango in the case of `P2PS`). 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 by probing periodically the neighbors, the routing table of a node in `P2PS` is corrected when the peers are actually using the network (as described in [11]). While this economic way is well suited for maintaining the routing table, it is 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 has failed, the peer has to refer to the next peer in the successor list), a peer should be notified immediately about all modifications of the r next succeeding peers (r is the length of the successor list).

4.3 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 three 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. Third are the monitoring primitives.

The application can interact with the Services layer by invoking the corresponding methods directly as well as by simply reading information on the two available input 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.

5 An Example Using P2PS

Here is a simple example of a P2P application composed of three peers that 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 an one-to-many message to `node1` and `node2`, and an one-to-one message to the responsible of key `42`, which will actually end up to `node1`. For more clarity, we purposely omitted the exception handling. The code runs directly in the OPI – Oz Programming Interface.

The first node of a P2P system is always “special”. Actually, it represents a system by itself. When creating a network (i.e., the first node) one can specify the network parameters. In our example, we decided to work with the default network values provided by the system. Nevertheless, we specify parameters for the node and its access point. That is, we indicate we want `nodeId=1` and that it should work on port number `3001`. Than, we run a loop over the message stream and displays the messages it receives. The following is the code implementing `node1`.

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

% Create the first node (with id 1) in the P2PS network.
OP2PS = {New P2PS.p2pServices
         createNet(nodeConfig: nodeConfig(nodeId:1)
                   apConfig:   apConfig(pn:3001))}

% Get the message stream and display each message received.
for M in {OP2PS getMsgStrm($)} 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, it runs a loop to wait and displays 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 an access point representation for the node to join to.
RAP = {P2PS.address2ap "127.0.0.1" 3001}
```

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

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. Further on, it sends an one-to-one message to the responsible of key `42` (which will actually end up 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']}

RAP = {P2PS.address2ap "127.0.0.1" 3002}

OP2PS = {New P2PS.p2pServices joinNet(remoteAP: RAP)}

{OP2PS send(dst:42 msg:anOzValue toResp:true)}
{OP2PS multicast(dst:[1 16] msg:hello)}
```

6 An Application Using P2PS

There are different applications that have been developed with `P2PS`. Some examples (renoir.info.ucl.ac.be/twiki/bin/view/INGI/Peer2PeerSystem) are PostIt, P2P-Matisse, and Community-Panel. In this section we describe the 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. Its main objective is to gather Oz developers concerned with a common problem in one community, and provide the community with tools for real-time collaborative edition.

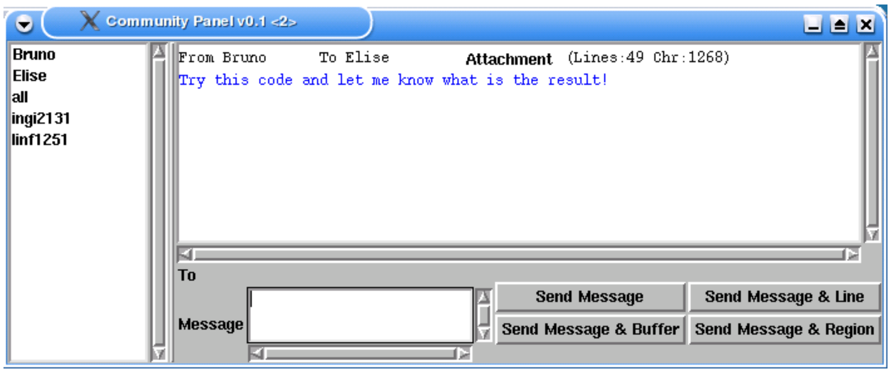

Fig. 2. The Community-Panel GUI.

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 at the user's request. Second, the application facilitates the communication between developers by supporting chat-like and instant messaging facilities.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.

From Figure 2 one can see that the GUI is composed of 3 areas: the membership, the received messages, and the submit. 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, just after the cursor's 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.

7 Conclusion

This document presents part of our ongoing work within the framework of extending Mozart/Oz with new programming abstractions to offer P2P abilities. Throughout the document we described `P2PS`, a P2P development platform for Mozart. We focused on its functionality and on its architecture as well as on how to write simple applications. The P2PS library is developed in Mozart/Oz and it implements Tango, a DHT-based algorithm. From its functionality, one can see that P2PS is simple to use and very practical to construct and work with large scale distributed applications; thus taking advantage of the provided P2P services and primitives. Furthermore, given the expressiveness of Mozart/Oz, we believe that P2PS is an attractive choice for developers.

The feedback – since one year now from its first release – we have been receiving 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 as well as to be extended with more specialized services. More encouraging, P2PS will be used as a distributed communication environment in further research projects at UCL and CETIC.

P2PS is the first Mozart/Oz development platform offering primitives for building P2P applications. It is delivered as a software package containing the source code together with an API documentation and an example-based user tutorial. Last fall we made the public release of P2PS on MOGUL (www.mozart-oz.org/mogul); the official archive of 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.

References

1. UCL, CETIC, Belgium (2003). P2PS: Peer-to-Peer System Library. http://www.mozart-oz.org/mogul/info/cetic_ucl/p2ps.html
2. Carton B, Mesaros V (2004) Improving the Scalability of Logarithmic-Degree DHT-based Peer-to-Peer Networks. In: Euro-Par
3. Stoica I, Morris R, Karger D, Kaashoek F, Balakrishnan H (2001) Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In: ACM SIGCOMM
4. Klintskog E, Mesaros V, El-Banna Z, Brand P, Haridi S (2003) A Peer-to-Peer Approach to Enhance Middleware Connectivity. In: OPODIS
5. Klintskog E, Brand P (2004) Extended Distribution Subsystem. D4.6 PEPITO deliverable http://www.sics.se/pepito
6. Dabek F, Zhao B, Druschel P, Kubiatowicz J, Stoica I (2003) Towards a Common API for Structured Peer-to-Peer Overlays. In: IPTPS
7. Rowstron A, Druschel P (2001) Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In: Middleware
8. Traersat B, Abdelaziz M, Pouyoul E (2003) Project JXTA: a Loosely-Consistent DHT Rendezvous Walker. White Paper, Sun Microsystems, Inc.
9. Russo F (2002) JXTAnthill. Master Thesis. Dept. of Computer Science, Bologna, Italy
10. El-Ansary S, Onana L, Brand P, Haridi S (2003) Efficient Broadcast in Structured P2P Netwoks. In: IPTPS
11. Onana L, El-Ansary S, Brand P, Haridi S (2003) DKS: A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications. In: CCGRID