



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

**Deliverable no: D5.3**

## **Report on Demonstrator Applications**

REPORT VERSION: first

REPORT PREPARATION DATE: 2004.12.31

CLASSIFICATION: Public

DELIVERABLE NO: D5.3      DUE DATE: Month 36      DELIVERY DATE: Month 36

PROJECT START DATE: 2002.01.01      PROJECT DURATION: 36 months

RESPONSIBLE PARTNER: UCL

PARTICIPATING PARTNERS: UCL, SICS, INRIA, EPFL

PROJECT COORDINATOR: Swedish Institute of Computer Science AB

PROJECT PARTNERS: UCL Louvain-la-Neuve, EPFL Lausanne, INRIA Paris, KTH Stockholm, 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: D5.3  
Report on Demonstrator Applications  
Due date: project month 36  
Delivery date: 2004-12-31

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

20th January 2005

Edited by Peter Van Roy

With contributions by Per Brand, Olov Ståhl, Anders Wallberg, Pär Hansson, Valentin Mesaros, Raphaël Collet, Celia Franco, Bruno Carton, Sébastien Baehni, Roland Flury, Rachid Guerraoui, and James Leifer.

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Demonstrator Summaries</b>	<b>5</b>
2.1	P2PmBlog Summary . . . . .	5
2.2	PostIt Summary . . . . .	5
2.3	Community Panel Summary . . . . .	6
2.4	Car Park Summary . . . . .	6
<b>3</b>	<b>P2PmBlog Demonstrator</b>	<b>6</b>
3.1	Background . . . . .	7
3.2	Peer-to-Peer Based Blogging . . . . .	8
3.3	The P2PmBlog System . . . . .	8
3.3.1	Architecture . . . . .	9
3.3.2	Applications . . . . .	11
3.4	Blog Node Services . . . . .	11
3.4.1	Query . . . . .	13
3.4.2	Subscription . . . . .	13
<b>4</b>	<b>PostIt Demonstrator</b>	<b>14</b>
4.1	Application Definition . . . . .	15
4.1.1	PostIt Functionalities . . . . .	15
4.1.2	Graphical User Interface . . . . .	15
4.2	Underlying P2P Platform . . . . .	17
4.2.1	Create a Network . . . . .	17
4.2.2	Join/Leave a Network . . . . .	17
4.2.3	Message Sending and Receiving . . . . .	18
4.2.4	Monitoring . . . . .	18
4.3	Application Architecture . . . . .	18
4.3.1	Functional Definition of the Modules . . . . .	18
4.3.2	Mapping Groups to P2PS Networks . . . . .	19
4.4	Conclusion . . . . .	20
<b>5</b>	<b>Community Panel Demonstrator</b>	<b>20</b>
5.1	Targeted Objectives . . . . .	20
5.2	System Architecture . . . . .	21
5.3	Using the Community Panel . . . . .	22
5.3.1	Entering the Community . . . . .	22
5.3.2	Community Panel GUI . . . . .	22
5.4	Working With a P2P Network . . . . .	23
5.4.1	Challenges . . . . .	24
5.4.2	Advantages . . . . .	24

<b>6</b>	<b>Car Park Demonstrator</b>	<b>24</b>
6.1	Prototype Overview . . . . .	25
6.1.1	Car Representation . . . . .	25
6.1.2	CarPark Representation . . . . .	26
6.1.3	SimC - The Simulation Center Representation . . . . .	27
6.2	Algorithms . . . . .	27
6.2.1	Gossiping the Availability of Free CarParks . . . . .	28
6.2.2	Counting the Cars in a Free CarPark . . . . .	28
6.2.3	High Availability of Paying CarParks . . . . .	30
6.2.4	Finding the Fastest Route to a Destination . . . . .	31
6.3	Design Overview . . . . .	32
6.4	Final Remarks . . . . .	33
	<b>References</b>	<b>34</b>

## 1 Introduction

The goal of workpackage 5 is to produce demonstrators for the testing and evaluation of systems and algorithms developed in workpackages 2, 3, and 4. We wanted to demonstrate the advantages of peer-to-peer technology, in general, and the advances made in PEPITO, in particular. Moreover, we want to demonstrate the technology in as tangible and easily grasped manner as possible - in order to promote dissemination by maximizing the target audience.

We developed the following four demonstrators that are described in this report:

- ▷ **P2PmBlog.** This is a distributed Weblog based on the DKS algorithm developed mainly by SICS. The software is released in deliverable D5.1.
- ▷ **PostIt.** This is a collaborative information-sharing application developed mainly by INRIA, in collaboration with UCL. The software is released in deliverable D5.2.
- ▷ **Community Panel.** This is a messaging service integrated into the Mozart program development environment. It is developed mainly by UCL, with some help from the CETIC laboratory [1]. The application and its user documentation are available at:

<http://renoir.info.ucl.ac.be/twiki/bin/view/INGI/CommunityPanel>

- ▷ **Car Park.** This is an information diffusion application intended to help cars find free spaces in carparks. It is developed mainly by EPFL. The application and its user documentation are available at:

<http://lpdwww.epfl.ch/sbaehni/work/carPark/carPark.html>

The software for two of these applications, P2PmBlog and PostIt, is released as deliverables D5.1 and D5.2. The other two applications are not project deliverables so we have made them available at the URLs given above. In the present report, Section 2 gives a quick overview of each application and the subsequent sections go into detail on several of the applications. Section 3 explains P2PmBlog, Section 4 explains PostIt, Section 5 explains the Community Panel, and Section 6 explains the Car Park.

The P2PmBlog, PostIt, and Community Panel applications all involve overlay (P2P) networks between end-user machines enabling or simplifying new forms of collaboration, rather than, say, overlays connecting servers together to achieve scalable, fault-tolerant (24/7) Web services. Additionally, we did not focus on demonstrating such properties as increased connectivity or the ability overcome NAT and firewall limitations (see D4.10). The chosen applications were, rather, chosen to rely as little as possible on improvements *under the hood*, but rather to maximize the perceived benefits for a wide audience.

The demonstrator applications make use of different systems developed in the PEPITO project. The P2PmBlog application makes use of the Distribution Subsystem (DSS) developed in WP4 and described in D4.10. The DSS in turns incorporates the DKS algorithm (or family of algorithms) developed in WP2. The PostIt and Community Panel applications make use of P2PS, a peer-to-peer library written in the Mozart Programming System [11]. It is based on the Tango overlay topology developed in WP2. Tango is a variation of DKS that improves scalability. The Car Park makes use of gossip-based broadcast algorithms that were developed in WP2.

**Dissemination** We plan to demonstrate several of the applications in the final PEPITO workshop and in the final review. The P2PmBlog application has been publicly demonstrated on numerous occasions in Sweden. In particular, it was demonstrated on SICS Industry Day in May 2004. The Community Panel was presented at Argonne National Laboratory (Chicago, USA) in June 2004. The P2PS library, which underlies the PostIt and Community Panel applications, has been released publicly in the MOGUL Archive, which is the third-party software library of the Mozart system [7, 11].

## 2 Demonstrator Summaries

This section gives a brief summary of each demonstrator application. They are explained in more detail in subsequent sections.

### 2.1 P2PmBlog Summary

The P2PmBlog application, released as deliverable D5.1, demonstrates collaboration between Blog creators and readers. From the system point of view Blogs are (for the most part) monotonically increasing (in time) data sets. They have been compared to diaries that are made publically available. Typically there is a small number of writers (contributors) and a large number of readers. This form of publishing information is, today, relatively common, but, up till now, the realization has all been server-based. The formation of peer-to-peer supported collaborations at the edge of the network for creators, contributors and readers of Blogs makes for a scalable, fault-tolerant, and self-managing Blogging infrastructure.

The application is based on a two level architecture in a novel way. At the heart of the application is an overlay network of nodes for storage, manipulation, and retrieval of Blog entries. Separate from this are the devices (e.g. mobile phone) or machines (e.g. PC) that interact with the overlay network. Any one user may have a number of such devices to access and make use of the Blog service that the overlay network provides. As the devices may have very different capacities the display of Blog entries can be adapted to the devices (in addition to the user). The separation between control device and P2P-node may be virtual; for example, a PC is often both control-device and P2P-node at the same time.

The application supports efficient querying services where the filtering is done at the source(s) not at the end-user (client) site. Users may subscribe to Blogs to receive notification of new entries. The associated prototype (deliverable D5.1) includes the peer-to-peer node code and several client device implementations (e.g., for mobile phones). The P2PmBlog application is more completely described in Section 3.

### 2.2 PostIt Summary

The PostIt application, released as deliverable D5.2, is a collaborative group-oriented information-sharing application. Within the overlay network formed by a large group of end-users, smaller groups may be formed. Each individual may be a member of arbitrary number of groups. The application uses the push paradigm. Messages posted to groups will be seen by all members of the groups without human or machine intervention (i.e. without polling). A *postit* is composed of two parts, one of which is modifiable only by the original creator, while the other is extensible by all members of a group. Postits have a limited lifetime as specified by the creator of the postit.

The PostIt application is more completely described in Section 4. Included in the description is also a short description of the P2PS library upon which the application is based.

### **2.3 Community Panel Summary**

The Community Panel is a messaging service that is integrated into the Mozart program development environment [2]. Messages typically contain both information designed to be read by program developers and code (or even data) that can be directly used in the programming development environment.

The Community Panel application also makes use of a Twiki Web server to facilitate easy entry into the various communities that are registered there. A more complete description of this application is given in Section 5. The Community Panel application is not a deliverable but is available at the URL given in the introduction.

### **2.4 Car Park Summary**

The Car Park application shows two different ways of finding car park in a city or a geographical region [3]. In the current implementation, a set of cars (which are running processes) drives in a probabilistic manner on the EPFL campus. The user can control one of them and can tell it where he wants to go. The car park algorithm then finds the nearest car park according to the user needs.

The current algorithm can be split in two distinct parts: (1) a paying scheme and (2) a free scheme. In the paying scheme, we assume that paying car parks disseminate the information of their number of free places throughout the campus. The cars receive that information and then can decide where to park. In the second scheme, the cars carry the information of the number of free places with them. When they meet each other, they transfer the information they have and update it. With this technique, the cars can approximate the number of free car parks places in the overall region and can get rid of the paying car parks.

The prototype lets us specify the total number of car parks (free or paying) and the total number of cars. Each car park and a car is modeled using a process and hence can be started on different computers. For convenient purpose, a graphical application has been implemented that monitors the total number of cars in the system as well as their current position. Another graphical application simulates the car that the user controls and with which he can specify where he wants to go. This last application shows dynamically the number of free car parks places.

The Car Park application is not a deliverable but is available at the URL given in the introduction. The Car Park is more completely described in Section 6.

## **3 P2PmBlog Demonstrator**

This WP5 demonstration activity uses the results of the PEPITO WP2 and WP4 workpackages to realize a sophisticated blogging service and to create a test application for peer-to-peer technology. More specifically, this is a demonstrator of the PEPITO work on DHTs and key-based routing algorithms (DKS/DSS) [10, 12]. The result is a blog system, called P2PmBlog, that uses the DSS platform [9] to store and distribute blog data among peer nodes, which can then be accessed by users via specific blog reader applications.

A blog is a set of Web pages, often maintained by a single person, where short messages of text and images are published on a daily basis. Many blogs are news or diary oriented and tend to be of personal nature. Today, blogs exist everywhere on the Internet and use standard Web technology (HTTP, HTML, XML) to represent and distribute content to end users.

The P2PmBlog system consists of P2P "blog nodes" sharing distributed data through the use of key-based routing algorithms. The P2P nodes are made up of three architectural layers, where the two lower ones are independent of the application-specific blog data knowledge. The lowest layer of each such node is made up of the DSS, incorporating the DKS algorithm. Compared to conventional (Web-based) blogging, the P2P solution gives high fault tolerance and availability at a low cost, and also allows subscription mechanisms to be used which decrease bandwidth consumption and host loads.

For demonstration and testing purposes the project has implemented a mobile J2ME midlet for mobile devices as well as a desktop application, which makes it possible for a user to conveniently browse blog content and generate new content. These applications do not themselves implement a P2P node, but accesses the blog data through a P2P node via http-based requests. Furthermore, external "blog extractor" processes can pull information from the regular Weblog systems and RSS sources on the Web into the P2P network, through any of the nodes, see Figure 1.

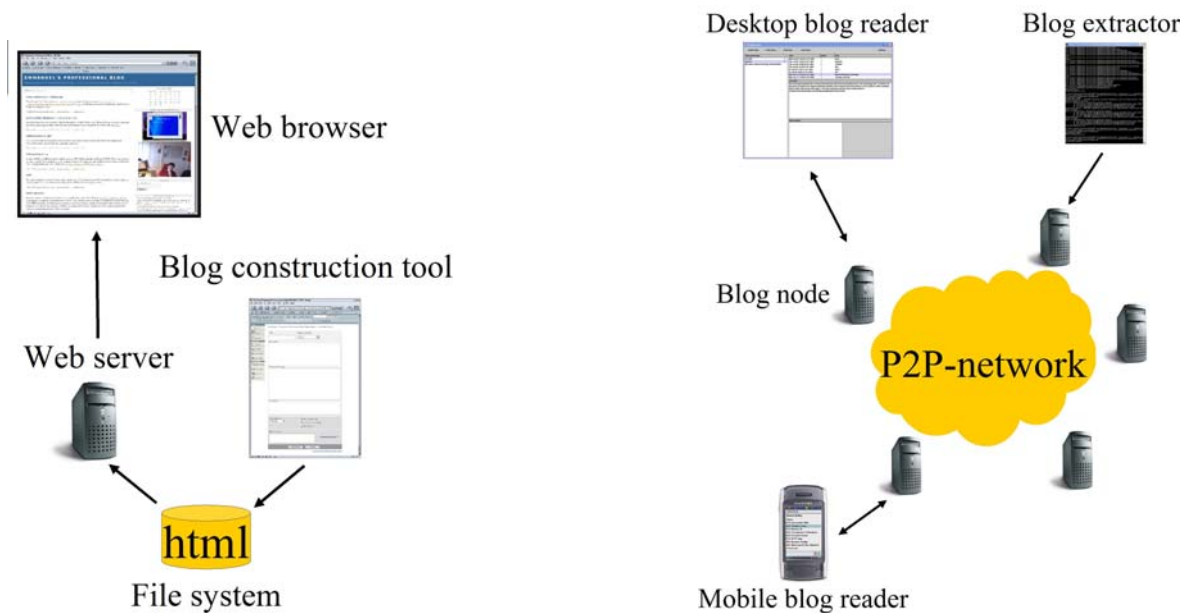


Figure 1: Web-based blogging vs P2P-based blogging

### 3.1 Background

Weblogs, or blogs in short, are a recent Internet phenomenon. Started as an underground culture in the late 90's, the movement has now reached such an extent that the content of these Web sites has become an important part of opinion building and information spreading on the Internet. The most famous example is the "Baghdad Blogger". Salam Pax was the pseudonym for a 29-year-old Iraqi architect whose online diary, featuring wry and candid observations about life in wartime, transformed him into a cult figure whose readership grew to millions, and his accounts were quoted in the New York Times, BBC, and Britain's Guardian newspaper. If the first Gulf War introduced the world to the "CNN effect," then the second Gulf War was blogging's coming out party. Salam Pax was the most famous blogger during that conflict, but myriads of other online diarists, including U.S. military personnel, emerged to offer real-time analysis and commentary.

Blogs are an easy way to produce a Web page, which is done by filling in simple forms. The text entered will arrive at the top of a page which is organized in a reverse chronological order. Blogs provide an archive mechanism for all entries and search features to look for specific information within older content. Typically, a blog will have a small number of authors and many readers. A commenting system will let readers react to the opinion expressed by the authors on the Web page.

The simplicity of the formula makes blogs available to the mass of Internet users, relieving them from the complexity of HTML editing. Apart from this simplicity, blogs are interesting from their sociological aspects and what authors are actually using them for. From that point of view, blogs have two main purposes: keeping together a community around any given topic (from a class trip to war in Iraq); and making one's opinions heard to the masses.

A Weblog system typically consists of Web server side software which delivers a simple forms-based user interface, for both posting and administering, through the Web. To manage all blog entry and meta data, the software is backed by a database system. Some systems generate dynamic Web pages, and other systems use a two-stage process where a set of static pages are regularly updated. Often these systems also supply additional interfaces, such as XML-RPC APIs, some of which are now used across several systems and have become de facto standards.

### **3.2 Peer-to-Peer Based Blogging**

Blogs have intrinsic peer-to-peer properties in that information is produced and consumed by users located at the edge of the network. In difference from traditional WWW models where information is produced by entities equipped with economical resources to maintain and sustain powerful Web servers, blogs are often maintained by individuals. Moreover, the contents of traditional Web pages have static characteristics, while the contents of blogs are subject to constant updates because of the diary property of the blogs. The architecture of choice for blogs have been the use of Web servers located in the (logical) center of the network.

The current technical limitation in blogs is the dependency on the single Web server that hosts a blog. The Web server becomes a single point of failure because of flash crowds, denial of service attacks, and network congestion. This is true even if the Web server is hosted at a machine under control of the blog author, thus at the edge of the network. We suggest using peer-to-peer techniques in order to provide a more reliable blog infrastructure that stores its information at the producers and consumers of blog data. Moreover, the similarities with how blogs are used and how file-sharing systems such as Gnutella, LimeWire, and Napster are used, is an argument for exploring the domain of blogs and peer-to-peer.

Blog systems based on peer-to-peer technology may also provide some more application related benefits to its users, in addition to the more system oriented ones like increased scalability and availability. For example, since all blogs are maintained by one system, it is possible for users to retrieve and browse a list of all available blogs. This may allow them to discover blogs they didn't know existed, and they may even be automatically notified when new blogs are created. This is such an important feature that servers have started to appear on the Web today where lists of available blogs are kept. However, this solution suffer from the same limitations as Web-based blogging in general, e.g., single point of failure, and only provide a small subset of the blogs existing on the Internet.

### **3.3 The P2PmBlog System**

P2PmBlog is a blogging system that runs on top of the DSS peer-to-peer platform. The system consists of blog nodes that may form large peer-to-peer systems that can be used to store and retrieve

blog data, and a number of *blog clients* that can connect to such nodes, for instance a reader that extracts and presents data to a user (see Figure 2). The connections are made using remote procedure calls, each node runs a minimal XML-RPC server that exports an API for posting and retrieving blog entries.

Each blog is represented in the system by a known key, typically the name of the blog. The XML-RPC API uses strings to represent keys, which are then hashed into integers by the blog node software before finally delivered to the DSS. If the blog also exists as a regular Weblog, i.e., on a Web server, the blog URL is used as the key. By using the key, RPC clients are able to post entries consisting of a collection of text and media objects (e.g., images). There is no limit on how many entries can be posted under the same key. When retrieving entries, a blog client specifies the blog key and will then receive sub-keys for all entries stored in the blog. The sub-keys can then be used to retrieve individual entries.

### 3.3.1 Architecture

A P2PmBlog node is made up of three different layers as seen in Figure 2. Each layer provides a well defined API which is used by the layer on top of it. The bottom layer consists of the DSS platform itself.

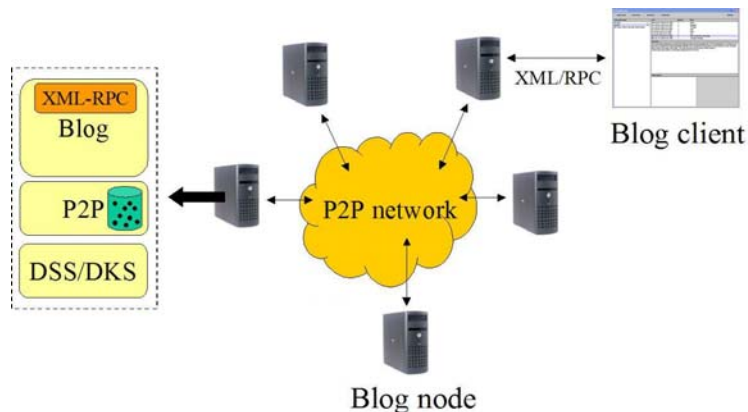


Figure 2: The P2PmBlog system architecture

The DSS is written in C++, while the two topmost layers are written in Java. JNI is used to interface Java to the C++ code of the DSS. The following sections will present each layer in more detail.

**The Blog Layer** The Blog layer provides methods for adding and retrieving blog information. This is the only layer of the P2P node that knows about blogs, entries, etc., and how these are represented by various Java classes. The basic services provided by a blog node (i.e., including the Blog layer) is:

- ▷ add and remove blogs.
- ▷ add and retrieve blog entries.
- ▷ add and remove subscriptions.
- ▷ perform queries on entries in blogs.

A user of the blogging system may have one, but also more than one, client for accessing blog information in the P2P network. Examples of blog clients are reader applications that may be run on a mobile phone or desktop computer. In the Blog layer, every client has its own set of preferences and subscriptions. In addition there are common user preferences shared by every client of the same user. Subscriptions are handled by the Blog layer for its clients using two mechanisms: clients are made aware of new entries posted to subscribed blogs either indirectly by asking for new entries, or by having the blog node push subscribed information via a socket connection. Subscriptions are described in more detail in Section 3.4.2.

Blog clients interface to the Blog layer via XML-RPC. XML-RPC is a popular protocol that uses XML over HTTP to implement remote procedure calls. Technically, this means that the Blog layer is capable of hosting a minimal server that exclusively handles XML-RPC requests. The XML-RPC handler more or less exports the features of the Blog layer API. However, there are some extra features in the handler that are added due to the limited channel and processing power of the clients. The XML-RPC handler provides several ways of retrieving more limited sets of entries in a blog. The Blog layer can also perform media format conversion to support devices with particular demands.

**The P2P Layer** The P2P layer provides an API that allows the Blog layer to store and retrieve arbitrary Java objects under string keys. The P2P layer uses the DSS/DKS layer (see next section) to route messages containing encoded objects to nodes that are responsible for the corresponding keys. The string keys are hashed into integers before given to the DSS/DKS layer.

Objects, once routed to the correct node, are stored in hash tables in the P2P layer. The Blog layer is able to store more than one object under any given key. In fact, each key is mapped to a dynamically sized array of objects. Each time the Blog layer stores an object, the object gets added to the array rather than replacing the current object (if any). Thus, the result of retrieving the data stored under a key is always an array. Whenever the Blog layer asks the P2P layer to retrieve the contents of a key, the P2P layer creates a message which is then routed to the node responsible for the key, using the DSS/DKS layer. The receiving node encodes the array of objects into a new message which is sent back to the original node, where the array is decoded and finally delivered to the Blog layer.

The P2P layer also provides a subscription API that allows the Blog layer to be notified whenever a new object is stored under a specific key. This mechanism is based on the publish/subscribe service of the DSS/DKS and generates a callback to the Blog layer whenever an object is added to the array of an subscribed key.

**The DSS/DKS Layer** The Distribution Subsystem (DSS), developed in WP4, is a middleware that provides a distribution service intended to extend programming systems with distribution support. Internally, the DSS implements an efficient messaging layer that can traverse fire walls and NATs, and handle migrating processes. On top of the messaging layer is a layer situated that implements various distributed algorithms, including the structured peer-to-peer system DKS from WP2. DKS is used as a basic service internal to the DSS, but also exposed outside of the DSS.

Delivered from WP4 is a library of data structures that can be used in a distributed setting, called C++DSS. This library was used to implement a generic key-value middleware. Using the DKS and the distributed data structures of C++DSS, a key-based routing service and a publish/subscribe service are provided.

### 3.3.2 Applications

Apart from the blog node software, P2PmBlog also consists of a number of stand-alone applications that corresponds to examples of different types of blog clients. These applications make use of the XML-RPC API which is provided by all nodes, and thus need not to run on the same machine as the node to which it connects. When these applications are started, they are given the address and IP number of a host where some P2P node process is running. However, since the connections between the applications and the P2P nodes are based on HTTP, applications can be reconfigured to connect to other nodes dynamically, i.e., without having to be restarted.

**The Blog Extractor** There is a blog extractor application in order to push external blog data into the P2P network, both for testing and real scenarios. An extensible framework for accessing different types of data sources exist, currently supporting:

- ▷ Blog systems with Blogger/MovableType/MetaWeblog XML-RPC interface
- ▷ RSS feeds
- ▷ Cache file generated from above sources

The blog extractor launches handlers for different source types, which periodically poll sources for updates and transforms data which is then sent to the P2P node using XML-RPC calls. Apart from the "live" sources, the application has the ability to write incoming session data to a cache file. This cache file can be useful for demonstration/test purposes without Internet access, where data can be read back as if coming from a live source at a predetermined rate. Update intervals, archive retrieval properties, and more, can be specified on a per-source basis through property text files.

**The Mobile Reader** A mobile blog client for a J2ME MIDP 2.0 environment has been developed in order to let users browse blog data in the P2P network using a mobile phone. The client connects to a "home" P2P node through the node's blog layer XML-RPC API over TCP/IP, using it as a proxy to access the P2P network. A user can browse blogs by going through a hierarchical GUI of blogs, blog entries, and blog entry content (see Figure 3).

The application also allows users to post new entries. Furthermore, some J2ME platforms support push, which the mobile reader utilizes to let the P2P node push information regarding new entries, which automatically launches the application if it is not running to show a list of subscribed blogs indicating newly arrived entries.

**The Desktop Reader** The desktop reader is another supported blog client type and provides users with a GUI that can be used to connect to the P2PmBlog system from an ordinary desktop computer, e.g., a PC. The reader GUI looks very much like a mail reader in appearance (see Figure 4), and allows the user to query which blogs exists in the system, to create new blogs, to subscribe to such blogs and to retrieve and read the blogs' entries. Users are also able to compose new entries which can then be posted to the selected blog.

## 3.4 Blog Node Services

Apart from the fundamental service of storing and retrieving blog entries, each blog node also provides a subscription service and a query service. Subscription allows clients to be automatically



Figure 3: The P2PmBlog mobile phone reader

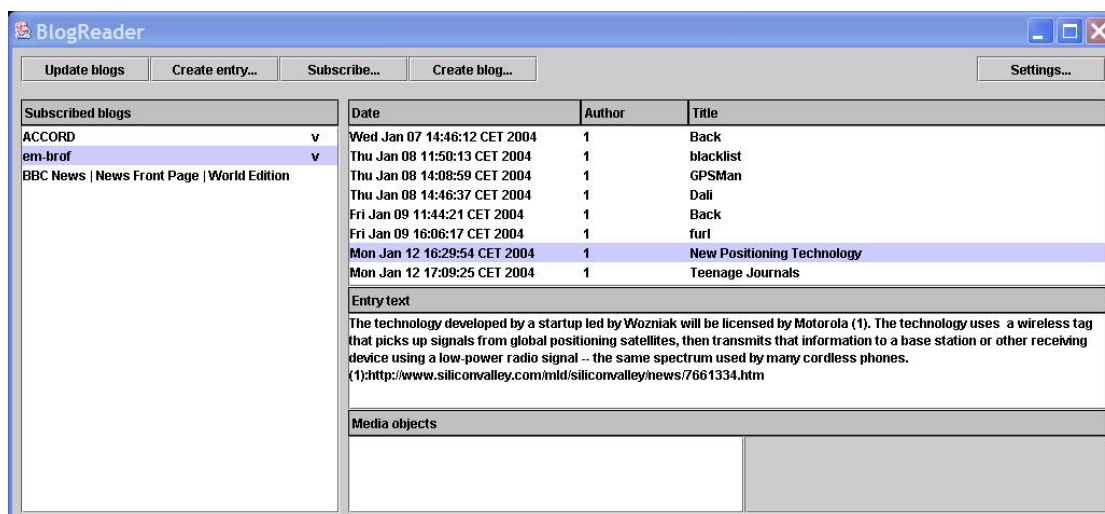


Figure 4: The P2PmBlog desktop reader

notified about blog changes via a push mechanism. The query service can be used to find blog entries that match specific patterns, e.g., a set of keywords. Each service is made available via the XML-RPC API. Both of these services could, to some extent, have been implemented in the client itself, but this would have been inefficient for various reasons which is explained in more detail in the following sections.

### 3.4.1 Query

The query service allows a client reader to find entries in a blog that matches a given pattern. Even though Weblog servers typically only display the five or so most recent entries, older entries are saved in a database and are accessible to readers via a search function. This is an important feature since most blogs are not just forums for presenting and discussing recent views or events, they also act as archives or records of old information where users may go to look for knowledge on some specific topic.

The query service in P2PmBlog is implemented as follows. A client reader initiates a query by sending a set of string keywords to a P2P node via the XML-RPC API, along with the name of the blog for which the query is valid. Instead of fetching all data that may match a query and performing the matching locally, the receiving blog node routes the query to nodes where the corresponding blog data is stored (based on the blog name key), in the form of a query expression. This is done by issuing a conditional get, i.e., a request to return data under the key only if it matches the expression. The query expression supplied in the conditional get is a generic object containing a query type identifier and the payload necessary to perform the matching. By distributing query expressions and performing the matching on each receiving node, bandwidth usage is minimized since only the blog data that actually matches the expression is sent back to the node performing the query.

Upon receiving data based on a query, the blog node combines all matching data from different nodes before supplying them to the requester, typically a blog client reader, for final presentation to the user. In the current implementation it is possible to perform queries for entries in blogs containing certain bits of text. In contrast to traditional server based blogs it would, using P2PmBlog, be possible to implement queries that span all existing blogs.

### 3.4.2 Subscription

Subscription is an important feature when considering blogs since it allows users to be notified automatically when a blog changes, instead of them actively having to check the blog for new entries. However, since the Web-based blog servers of today doesn't typically support subscription, RSS readers and the like tend to implement subscription via polling, if at all. Such readers repeatedly (e.g., once every fifteen minutes) request a list of current entries from the blog server, and then compare the result to the list which was returned on the previous call. If the lists differ, new entries have been posted and the user is notified. This solution has several potential drawbacks. For instance, if the polling is done frequently, bandwidth will be wasted since there will be few or no changes between calls. Frequent polling will also increase the load of the blog server. If the polling is done infrequently, users will be notified about new entries long after they have appeared on the blog.

The P2PmBlog system provides a subscription mechanism that allows a blog client application, e.g., the mobile reader, to be notified each time a new entry has been posted to a specific blog. This mechanism is based on the publish/subscribe service of the DSS/DKS which allows for notifications when data has been stored under a specific key. When data is added to a key, all nodes that subscribe to this key will be notified automatically by the DSS via a callback invocation. The blog client

applications responsible for activating the subscriptions are then notified by the nodes via a push mechanism (a TCP socket is created from the node to the blog client) if supported. If the blog client doesn't support push (e.g., some mobile phones might not), it needs to check for new entries by periodically calling one of the XML-RPC API functions.

All subscriptions are persistent, which means that they are valid until explicitly removed by the client applications. As long as the subscription is not removed, the node will monitor the corresponding keys and notify the client whenever new entries appear. A problem occurs however if a client reader connects to a different node at some later point in time. The reason for this is that the DSS subscription mechanism is node based while the subscription service exported by P2PmBlog via the XML-RPC API is client based. Thus, on the DSS level the new node is unaware of any subscriptions that the client may have registered on some other node. This behavior will most certainly be confusing to the user. For example, if the user wants to remove a subscription (via the client reader GUI), the client reader may display information that indicates that the user is not actually subscribing to that particular blog. In order to remove the subscription, the user must first configure the client reader to connect to the node where the subscription was initiated. This problem will be fixed in future releases of the P2PmBlog system, until then users are recommended to configure their client readers to always connect to the same blog node.

A solution to the subscription problem would be to store a subscription state for each client application in the peer-to-peer system itself, and then have each node retrieve this state when a client connects. This would also require all nodes which have active subscriptions initiated by that client to subscribe to changes to this state in order to handle unsubscriptions. In this way a client could connect to any node which would then retrieve the state and thus know which blogs the client is currently subscribing. If the user then deactivates a subscription, the current node would change the subscription state, which would trigger a callback on all nodes subscribing to state changes. These nodes would then have to check the state and remove subscriptions which the user no longer wants to be active. The subscription state could be stored under a specific key which would be unique for each client, for instance based on some kind of identifier which the client supplies when it connects to a node.

## 4 PostIt Demonstrator

Distributed collaborative applications became part of our everyday life, but the development of such applications is still difficult. Robustness is hard to achieve, and efficiency is often poor. The classical client-server architecture revealed to be inappropriate in general for such applications. Advances in peer-to-peer computing give more insights on how to deploy applications that achieve better efficiency, and do not rely on single points of failures. This application uses the peer-to-peer library P2PS, developed in WP3 in Mozart by UCL and with some help from the CETIC laboratory, which is interested in exploiting the results [14, 11, 1]. The P2PS library is available in the MOGUL Archive [7].

PostIt is a distributed collaborative application that exploits the functionalities provided by P2PS. Users of PostIt share a virtual board for publishing short-lived texts. Each message is addressed to a group of users, and has a limited lifetime. We use the P2PS library as a group communication primitive, as it implements an efficient broadcast mechanism among the nodes of the peer-to-peer network. We also use the *lookup* facility in order to provide fast access to shared information.

To summarize, this section explains *how* our application uses P2PS to achieve its aim, and *why* the library is appropriate for such an application. Section 4.1 describes the application in general, its goals and user interface. Section 4.2 presents the P2PS library, and Section 4.3 gives our application's

architecture, together with insights on how we implement the functionalities on top of P2PS.

## 4.1 Application Definition

The main functionality of our application is to allow groups of users to communicate. The users exchange short messages with a limited lifetime. Messages can be sent to one or a subgroup of users. This section first defines more precisely the functionalities of the application, then describes its user interface.

### 4.1.1 PostIt Functionalities

The PostIt application considers three kinds of entities, namely the user, the group of users, and the message.

**A user** is identified by a *name* in the application.

**A group** is a set of users. Each group is itself identified by a name. A given user may be member of several groups. The name of a user can be thought of as a group containing that user only.

**A message** is a short text sent by a user to another user or a group of users. A message is also called a *postit*. The message is given a duration by its sender, after which it is *expired*. Before this happens, the message is said to be *active*. Messages are composed of two parts: the sender, destination, and duration form the *header* of the postit, while its *body* is given by the text.

We now define how these entities interact with each other.

- ▷ A user can send a postit to another user or a group of users. In the latter case, the sender must be member of the group to which the postit is sent.
- ▷ A user can change the contents of a postit. The header can be changed by the original sender only. The body can be extended by any user who has received the postit. So one can only add some text, no deletion or change is possible. Body extensions are not ordered.
- ▷ The user must be informed of the sender, destination, body, and remaining duration of each postit before it expires. The user only sees postits of groups he is part of.
- ▷ When a user joins a group, he should receive a copy of all active postits in this group.
- ▷ Active postits should be saved in a file. This will give some robustness to the application.

### 4.1.2 Graphical User Interface

We describe here the windows that show the currently active postits, notify postit arrivals, create new postits, and configure various options.

Figure 5 shows the main window. It presents a list of the active postits. The postit creation window can be incorporated in the main window if the users decides it. The main window provides three menu items, named *File*, *Edit*, and *Help*. The most important is *File*, which contains the main functionalities of the application.

**Create** opens the window for creating a new postit.

**Save** saves all postits received so far. Available only when the corresponding option below is active.



Figure 5: The main window, with the creation window included.



Figure 6: The postit creation window.

**Mask** hides a postit whose body is currently being shown. The hidden postit is still visible in the active list.

**History** shows all the postits received so far. This entry is available only when the corresponding option below is active.

**Group** allows the user to create a new group, join an existing group, or leave a group he is currently part of.

**Configure** allows to setup various options, such as:

- ▷ the message arrival notification, which may appear as a popup window, or a beep, or simply by a visual differentiation in the message list.
- ▷ the presence of the creation window inside the main window.
- ▷ whether all received postits are kept (for “Save” and “History” above).

The postit creation window is shown in Figure 6. The user enters the destination, either a group, or a user; the message lifetime; and the message body.

Figure 7 shows the reception window. It is composed of the message header, which is modifiable by the postit’s creator only, and the message body, including all the message extensions. Since the message header cannot be modified by all users, we provide two variants of the modification window: the postit’s creator window (see Figure 8), and the other users’ window (see Figure 9).



Figure 7: The reception window when the “popup” option is active.



Figure 8: The modification window for the user who sent the original postit.



Figure 9: The modification window for the other users.

## 4.2 Underlying P2P Platform

In this section we present the main abilities of the P2P platform, called P2PS [14], that we use as underlying infrastructure for our application. 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 P2PS tutorial [14].

P2PS implements Tango [6], a peer-to-peer algorithm for structured P2P systems. Unlike unstructured P2P systems like Gnutella (gnutella.wego.com), whose overlay topology is ad-hoc, structured P2P systems organize their overlay by following well specified rules in order to improve overall efficiency. Tango provides overlay topology maintenance, self-organization, efficient data lookup, and fault-resilience. Tango is based on the idea of Distributed Hash Table – DHT, where key#value pairs are associated to nodes in the overlay network depending on the “distance” between the key *id* and the nodes’ *ids*. Moreover, in order to achieve efficient data lookup, each node is connected to other  $\log N$  nodes in the overlay network, where  $N$  represents the maximum number of nodes in the network. Tango is closely related to DKS [?] but significantly improves scalability.

The main functionality of the P2PS platform 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.

### 4.2.1 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 access point is created for this node. An access point represents an addressable entry point of a node. The access point can be published, thus allowing other nodes to connect and join the overlay network. Furthermore, at initialization a peer 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.

### 4.2.2 Join/Leave a Network

When joining an overlay network, a peer node  $n$  needs to have the knowledge of an access point 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  $p$ . The system will self-organize in order to guarantee

overall efficiency and to remain resilient to node failures. 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.

Leaving an overlay network means implicitly disconnecting this node from all the other nodes it is connecting to in the overlay network. Underneath, a node will run a simple protocol to disconnect it from its neighbors. This will terminate the message and the event streams.

### 4.2.3 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, though the reliable send can be made synchronous. The provided communication primitives are: one-to-one best-effort and reliable, one-to-many best-effort: multicast and broadcast. Note that the multicast is explicit, i.e., one has to explicitly specify the destination nodes. 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 the associated message stream.

### 4.2.4 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. P2PS provides 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. Another way of monitoring the peer node is offered in the form of counters. For example, one can have information about the following: 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.

## 4.3 Application Architecture

The architecture of PostIt consists of three layered modules: the Graphical User Interface module (GUI), the Group Communication module, and the P2PS module. Each module only interacts with the modules right above or under it.

### 4.3.1 Functional Definition of the Modules

**The GUI Module** This module mainly interacts with the Group Communication module to send and receive messages. Received messages are notified and shown, depending on the user's options. Messages are sent by the postit creation window. Other interactions are related to group management, when the user creates a new group or joins an existing group.

**The Group Communication Module** This module is composed of the message manager and the group manager. The message manager contains a reception agent and a dispatch agent. The reception agent receives messages form the underlying layer and transmits them to the GUI. The dispatch accepts messages from the GUI, and queries the group manager in order to decide where to send it. The group manager basically consists in a table, that maps each group the user is part of to a

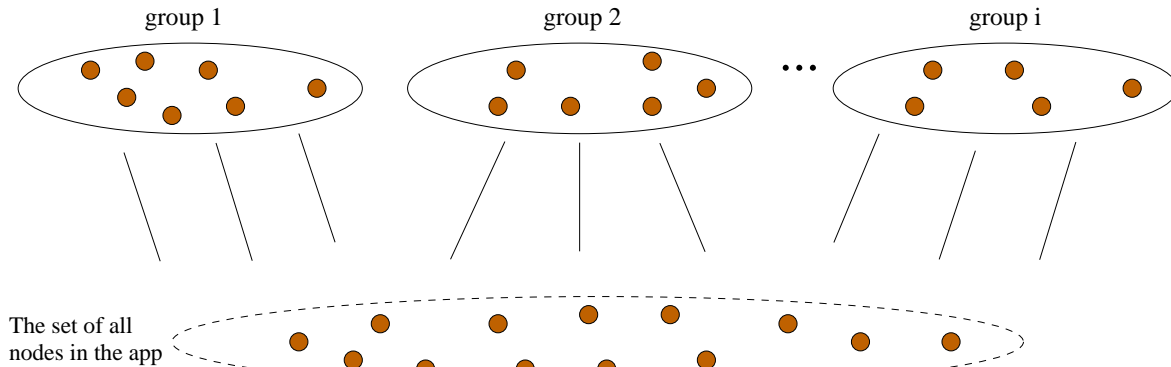


Figure 10: Groups as P2P networks.

communication channel. The table is updated when a new group is created, and when the user joins or leaves an existing group.

**The P2PS Module** This module provides the communication primitives for the application. It allows to create a P2P network, join a P2P network, etc. We mainly use two communication primitives: one-to-one message send, and one-to-many broadcast send.

#### 4.3.2 Mapping Groups to P2PS Networks

In this section we describe how we intend to use P2PS in order to form groups within PostIt. In PostIt we want to create groups that can be addressed by their names and where the group communication is efficient.

Recall that P2PS already provides group communication primitives: explicit multicast and broadcast. Nevertheless, while these primitives will be useful for our application, we still need more. That is, we need an implicit multicast method, where sending a message to a group does not require the knowledge of all members of that group. To this end, we chose to have a P2P network per group, where a node in a P2P system represents a PostIt process associated with a user. Moreover, a user (and consequently, the associated peer node) can be part of different groups. The same idea was proposed in [13].

Joining a group requires to find a P2P access point. It can be found by external means, for instance a Web page updated manually, or automatically by the users. Such an external source is necessary for the very first connection of a user. Once a user is connected, he can find access points for other groups via the application itself. For this it suffices that all users form one global group, like suggested in Figure 10. The access point can then be found by performing a lookup inside the global P2P network.

When a user joins a group, the application signals his presence by broadcasting a control message in the group's network. We propose to maintain the list of users of a group by a keep-alive mechanism. Every member of a group regularly broadcasts a control message in the group's network. A user from which no message has been received for a long period of time is considered to have left to group.

## 4.4 Conclusion

We have designed the collaborative application PostIt to be developed on top of a peer-to-peer network. The design is not specific to any peer-to-peer system, but the application's requirements make the P2PS library quite appropriate for it. We have shown how to use P2PS to implement PostIt. The application's main functionality is to send messages to groups of users. P2PS provides enough flexibility to create as many P2P networks as groups. This allows to send messages to a group without having to know all the users in this group.

## 5 Community Panel Demonstrator

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 group of people, not necessarily located in the same place. Collaboration is a time-consuming activity. 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 takes place, why not including collaborative code edition capabilities alongside the editor, compiler and debugger?

For our first step toward a collaborative IDE, we developed an application that we called *Community Panel* [2]. Its main objectives are to gather Oz developers concerned with a common problem in one community, and provide with tools for real-time collaborative edition.

In order to take advantage of various functionalities of a peer-to-peer (P2P) system, we implemented the Community Panel on top of our peer-to-peer library, P2PS [14]. Therefore, given the inherent scalability of a structured P2P system, the Community Panel can support multiple groups working within the same system, while providing them with efficient communication primitives.

The remainder of this section is organized as follows. We start by enumerating the targeted objectives. Section 5.2 describes the system architecture and how its main components interact with each other. Section 5.3 gives a brief description of how one can use the Community Panel. Finally, Section 5.4 describes the particularity of working on top of a structured peer-to-peer network.

### 5.1 Targeted Objectives

The targeted functionalities of the Community Panel are threefold.

First, the application provides users with community (e.g., group of developers) membership information. This information can be partial or complete, regarding the size of the community and scalability issues, but it 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 interact with the appropriated community or person.

Finally, the Community Panel provides a developing framework for exchanging code in text/binary format, and also language entities. For instance, one can imagine to develop an application by adopting a component based architecture where the Community Panel would play the role of real-time component connector.

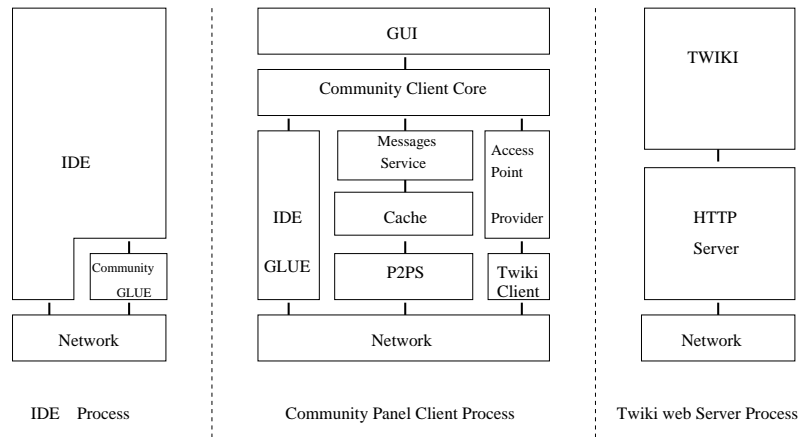


Figure 11: Community Panel software architecture.

## 5.2 System Architecture

The Community Panel is a distributed application involving three kinds of processes, as shown in Figure 11: the Community Panel Client, the Integrated Development Environment (IDE), and the Twiki Web server.

The role of the Twiki Web server is to act as a yellow page, storing entry points to the network formed by the processes currently running Community Panel Client (the so called Access Points, containing an IP address and a Port Number). The role of the IDE process, beside its conventional help to software development, is to provide the Community Panel Client with the ability to obtain some part of a program text as well as to insert program instructions inside the program text. Finally, a Community Panel Client interacts with other Community Panel Clients through the P2PS library in order to exchange messages and to maintain membership. Moreover, the Community Panel communicates with the IDE through the IDE GLUE and with the Twiki Web server through the module chain Access Point Provider and Twiki Client.

In the following we concentrate on the module composing the Community Panel Client in a top-bottom fashion. The graphical user interface (GUI) allows the user to connect to and disconnect from the Community Panel network. No intelligence is placed inside the GUI. All its functionality relies on the Community Client Core module which combines the functionality provided by the IDE GLUE, the Messages Service, and the Access Point Provider. In order to have a loosely coupled architecture, we designed our modules to communicate by exchanging events between each other.

**Graphical User Interface** The task of the GUI module is to display graphically to the user the membership of the Community Panel Network and the received messages as well as to provide access to the functionalities offered by the Community Client Core.

**Community Client Core** The Community Client Core has three functionalities. First, it allows to join and leave a Community Panel Network, based on a URL referencing a Twiki page. Second, it sends to and receives from the Community Panel Network messages with code attachment. Third, it provides membership notifications.

**IDE GLUE** IDE GLUE is an adaptor between the Community Client Core module and the IDE. It proposes a standard interface allowing the Community Client to obtain the text edited in the IDE as well as inserting text inside an IDE buffer.

**Access Point Provider** Given a URL, the Access Point Provider can return a list of access points contained in the Web page referenced by the URL and it can also insert access point descriptions into a Web page. In order to communicate with the Twiki server, the Access Point Provider module relies on the Twiki Client.

**Message Service** The message service maintains the membership of the Community Panel Network by relying on P2PS events which are relayed by the Cache module (i.e. `new(succ:SuccId)`, `new(pred:PredId)` and `new(succlst:SuccLst)`). Beside the membership, the Message service offers a notification mechanism when a message is received and also allows to retrieve the received messages and the attachments from the Cache.

**Cache** The Cache has two tasks : store the received messages and attachments, and provide communication reliability. The Community Panel was designed and implemented when P2PS did not offer any communication reliability. The Cache extends the P2PS broadcast in order to provide a reliable broadcast. This is done by archiving temporary received messages and by synchronizing the Cache when disturbances occur within the P2P network. These disturbances are indicated by P2PS events such as `new(succ:SuccId)`, `new(pred:PredId)`, and `new(succlst:SuccLst)`.

### 5.3 Using the Community Panel

The Community Panel is based on the P2PS library, and it offers to the Oz programmer with the possibility to communicate with each other. The current version is a first attempt at a *collaborative IDE*. The current implemented functionalities are chat-like communication, augmented with Oz source code sharing. In the next versions developers will be able to share Oz code (i.e., text) and Oz language entities (language references and values, e.g., first-class software components, called functors).

#### 5.3.1 Entering the Community

The first step one would do when using the Community Panel, is to register, providing a user name, and groups to be involved in. Still, before being able to collaborate with other developers, one has to reach them. To this end, the Community Panel uses a Web cache (a Twiki server) as a repository of access points to groups of developers.

#### 5.3.2 Community Panel GUI

As shown in Figure 12, one can see that the GUI of the Community Panel is composed of three 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 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 source code will be inserted in the current buffer of the OPI, just after the position of the cursor.

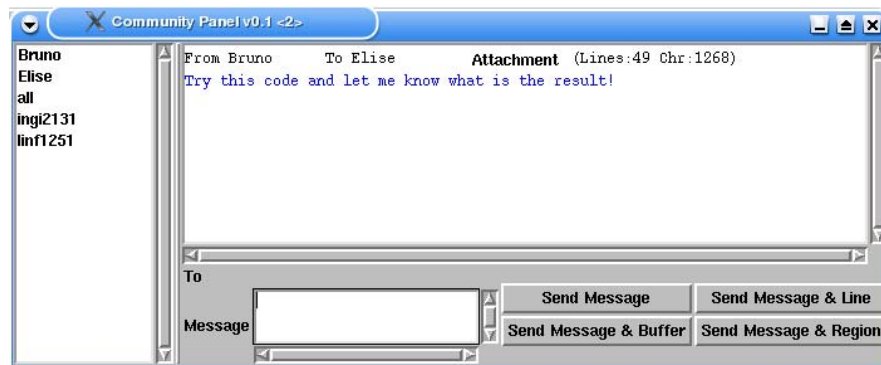


Figure 12: The Community Panel GUI.

### Sending a Message

The submit area is composed of a text box allowing to write a message, and four buttons to submit a message:

**Send Message.** Send a message containing the text from the text box to the users selected from the membership area.

**Send Message & Buffer.** The same as for *Send Message*, with the difference that the whole content of the current emacs buffer (i.e., the OPI) is put as attachment of the message.

**Send Message & Region.** The same as for *Send Message*, with the difference that the content of the selected region from emacs is put as attachment of the message.

**Send Message & Line.** The same as for *Send Message*, with the difference that the content of the current line in the emacs buffer is put as attachment of this message.

### Retrieving a Message

Once the user received a message with an attachment, he can retrieve the corresponding Oz-Code by clicking on Attachment onto the received message area. The retrieved source code will be inserted in the current emacs buffer, just after the cursor position. From this point on, the developer can immediately run the received code (alone or within the framework of an application), and eventually respond with some remarks or suggestions to the source code author.

## 5.4 Working With a P2P Network

Given its dynamic topology, a P2P system is known to be tricky to work with. However, a P2P system provides nice properties such as decentralization, scalability, and availability. In this section we briefly describe the particularity of working on top of structured P2P networks, such as Tango, Chord, and Pastry. We enumerate the main challenges one should take, as well as how one could exploit some of the functionality provided by a P2P network.

### 5.4.1 Challenges

**Dynamic Topology** In a P2P system nodes join and fail with a relatively high rate. Thus, a P2P system should inherently deal with these failures, and adapt itself to provide efficient communication. The Community Panel has to take this dynamics into consideration, too. As the P2P network evolves, the Community Panel has to follow it. We do this by synchronizing on each node in the network with its main neighbors (e.g., its successor).

**Multihop Communication** While in a classical client/server model the communication is done point-to-point, in a P2P system the communication between two nodes can pass through up to a maximum  $R$  number of other nodes within the system. Where  $R$  is in relation with the maximum number of nodes in the system. Not having control of the nodes that would forward the messages from a source to a destination can raise security and reliability problems. When developing the Community Panel we were not worried about the security problem. For the communication reliability, we implemented our own mechanism to recover from message failures. However, as P2PS has begun providing reliable send primitives, we intend to use them.

### 5.4.2 Advantages

**Network Scalability and Efficient Communication** A very useful functionality of a P2P system is the efficient communication primitives that it can offer. For example, in a P2PS network of  $N$  nodes, a message from any node  $n$  to any other node in the network  $m$  will take at most  $\log N$  hops. This nice functionality is due to the inherent scalability of the structured P2P systems. Beside end-to-end communication primitives, most of the structured P2P systems also offer group communication primitives such as broadcast and multicast. For the Community Panel we mostly used the broadcast primitive.

**High Availability** Since within a P2P system there is no centralized site controlling the well function of the system, the problem of the single node of failure is inherently solved. Moreover, a node within a P2P system can serve as an entry points to the system. For the Community Panel we use a Web server as a repository of access point to the Community Panel. Any node can publish its access point to the server, and eventually will be used as an entry point to the system by another node. Thus, there will exist several access points to the same network. Having multiple entry points to the system together with the network self-organization, makes a P2P system to be considered highly available. We exploit this functionality in the Community Panel.

## 6 Car Park Demonstrator

Today, GPS-based systems guide cars through cities to car parks without having any information about the availability of parking lots in the car parks. If no parking lots are available, the driver has to look himself for another parking facility nearby. Modern car parks are even equipped with a computer that counts the number of free parking lots in the car park. In some cities, this information is displayed at several locations to help drivers find a car park with available parking lots in a reasonable time.

It is however not reasonable to assume that all car parks are equipped with the infrastructure needed to detect their availability and then broadcast this information. This is especially true if we allow parking lots along the streets to be car parks (with a single place). This observation led us to explore an approach with two types of car parks. The first is the so called *paying* car parks, which

are equipped with all the technology needed to broadcast their availability. The name comes from the idea that this information might only be obtained against payment. The second type is called *free* car parks. They are not equipped with anything: it is up to the cars to detect and spread the availability of those car parks. Those free car parks will include single parking lots along the streets. The name *free* means that the information about the availability can be obtained for free.

In this prototype, we have developed the idea of displaying the availability of car parks in a geographical region. We assume that each car is equipped with GPS and wireless communication technology that allows to communicate with nearby receivers<sup>1</sup> in a Peer-to-Peer (P2P) fashion. With this equipment, the cars can easily gossip the availability of car parks from to nearby cars allowing them to construct a view of the availability of the car parks in the geographical region they are driving on. The wireless devices make it possible for the cars to receive broadcast information from paying car parks. With this information, the routing algorithm can dynamically adapt the route to the closest car park that has free parking lots. The algorithms used in the prototype are based on the papers [8, 5, 4].

This section is structured as follows. Section 6.1 presents a brief overview of the simulator. Section 6.2 presents the algorithms used in our distributed car park prototype. Section 6.3 overviews the design implementation of our prototype. Section 6.4 presents several remarks and gives pointers to more information about the Distributed Car Park Prototype.

## 6.1 Prototype Overview

The simulation takes place in a environment described by a *map*. Currently, EPFL with its major car parks is used for this purpose. The application consists of three major components, namely the cars, the car parks (referred to as CarPark), and the simulation center (often abbreviated with SimC). The following 3 subsections describe the task of those components. This Section gives an overview of what the simulation does and how those tasks are distributed to the different components.

### 6.1.1 Car Representation

A car is an entity that moves from an *entry point*<sup>2</sup> to a sequence of CarParks and finally leaves the simulation environment.

The driver of the car can specify a target point (not necessarily a CarPark). The car will then look for the closest<sup>3</sup> CarPark and calculate the fastest<sup>4</sup> route to this CarPark. While driving to the CarPark, the car receives information about the availability of CarParks, and, if necessary, changes dynamically its route to another<sup>5</sup> CarPark.

Arrived at a CarPark, the car looks for a free parking lot. If it finds one, it parks. Otherwise, the car assumes that the CarPark has no available places, remembers this information and goes to the next closest CarPark.

On the road, cars gossip the availability of free car parks they are aware of. Nearby passing cars receive this information and update their information if the received info is more recent. Note that information about paying CarParks is not broadcast with those update messages sent by cars.

---

<sup>1</sup>Among others, WLAN 802.11 or Bluetooth could be used.

<sup>2</sup>An *entry point* is a location on the map where cars can enter (and also exit) the simulation environment.

<sup>3</sup>The Euclidean distance to the center of the CarParks is used as a measure.

<sup>4</sup>Streets can be assigned different speed limits.

<sup>5</sup>CarParks are chosen in increasing order of the distance to the target point.



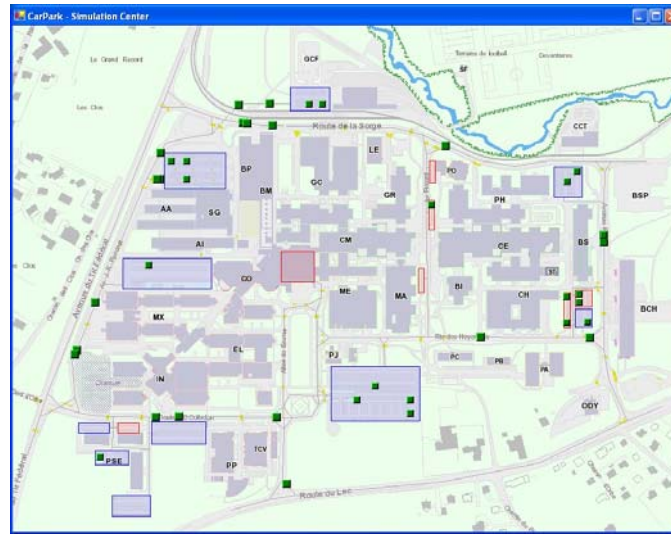


Figure 14: The simulation center GUI.

be the car that asks the barrier to open, in reality, it might be a sensor in the CarPark that detects the presence of the car.

Each paying CarPark (that is up, i.e. has a working computer) broadcasts its availability. All cars that can park in paying CarPark receive those updates, independent of their relative location. For the free CarParks, the information about the availability is broadcast indirectly via the cars: Whenever a car leaves the CarPark, it determines the number of cars that are parked in the CarPark<sup>6</sup>. It deducts the number of free parking lots in the CarPark and broadcasts this information to all nearby passing cars.

### 6.1.3 SimC - The Simulation Center Representation

The simulation center (SimC) is a utility of this simulation and does not correspond to any physical object in reality. Its main purpose is to display an overview of the system with all cars (see Figure14 for a screenshot of its GUI).

The SimC performs general tasks such as sending time synchronization messages, distributing unique IDs for cars and CarParks, and it keeps track of the address (IP:port pair) of CarParks.

## 6.2 Algorithms

This section contains some comments about the algorithms used in the distributed car park prototype. We first focus on the dissemination of the availability of the free car parks, then present the decentralized algorithm for counting the number cars in a free car park, then show how to achieve high availability in paying car parks and finally present the algorithm for routing the cars to their correct destination in the fastest way.

<sup>6</sup>Counting the number of cars in a CarPark is done during the whole time the car is parked. (See section 6.2.2.)

### 6.2.1 Gossiping the Availability of Free CarParks

Reliable information about the availability of a free CarPark is obtained when a car leaves a CarPark. During its stay, the car has run the algorithm to obtain the number of cars in the CarPark (see Section 6.2.2). While leaving, this car can deduct the number of free places, knowing the total parking lots of the CarPark. It stores the number of available places together with the time of the CarPark, which is obtained during the sign-off handshake (over TCP).

Each car holds a list in which it stores the availability of free CarParks. (Actually, this same list is also used for the paying CarParks.) The list might look like

CarParkID	free	timeStamp
1	0	13:45:53
3	4	14:23:44
4	2	08:22:48

where the first column contains the (unique) ID of the CarPark, the second the number of free places and the third a time stamp. The time stamp specifies when this information was created by a car leaving the CarPark.

Periodically, each car gossips this list to all its current neighbors (the period can be specified in the car configuration file). Before sending an update message, the car prunes out stale information. This means that it forgets entries whose time stamp is too old, compared to the local time of the car. As already discussed earlier, this step requires a (light) time synchronization of all cars and CarParks.

Upon receipt of such an update message, a car accepts entries that are new (no entry for the CarPark) or entries for which the time stamp is more recent than the current entry for this CarPark. Entries with stale information are denied, even if they are new.

For this application, only update messages that are sent from a car, whose location is within a certain distance to the receiver car, are accepted. This simulates the limited range of the wireless technology. Again, this range can be specified in the car configuration file.

### 6.2.2 Counting the Cars in a Free CarPark

The cars parked in a free CarPark need to determine the number of free places in the CarPark. The chosen algorithm determines the number of cars parked in the CarPark and then deducts the availability, knowing the total number of parking lots.

Each car has a unique ID (obtained from the SimC during the sing-up process) and is located at some fixed point in the CarPark. Cars can join or leave as they please, maybe even change the location. Cars can communicate with other cars via a multicast medium within a limited range. Each car has the task to determine, in a P2P way, the number of other cars that are parked in the same CarPark.

In our algorithm, each car keeps a list of the following columns which entries are, for example:

CarID:int	hops:int	ReceiveTime:time	Forward:bool
1	1	13:24:45	false
3	1	13:24:46	true
4	2	13:24:46	true
5	2	13:24:44	false

The CarID is the ID of the car for which this entry stands for, the hops specifies the number of other cars that forwarded this information before it arrived at this car, the ReceiveTime is the local

time of this car when it received this info and the `Forward` indicates whether this info has to be sent in the next update message sent by this car.

Periodically<sup>7</sup>, each car sends an update, always containing its own ID with the hops-value set to 0. This message also contains all entries from the list where the `Forward` bit is set to true. Note that those update messages only contain the car-ID and the hops info, but not the local receive time. After sending an update, all `Forward` bits in the list are reset to `false`. Assuming that the car that owns the list shown above has ID 9, an update message looks like:

carID	hops
9	0
3	1
4	2

Upon receipt of an update list, a car accepts an entry ( $ID_e:hops_e$ ) from the update if

- ▷ The car's list does not contain an entry for  $ID_e$ , and  $ID_e$  is different from its own ID
- ▷ The car's list contains an entry for  $ID_e$ , but  $hops_e$  is *smaller*<sup>8</sup> than the current hops value.
- ▷ The car's list contains an entry for  $ID_e$ , but the receive time of the current entry is too old, i.e. if  $ReceiveTime + timeout$ <sup>9</sup> is less than the current time.

If the update entry is accepted, a new entry of the form

$(ID_e, hops_e + 1, currentTime, true)$

is added to the car's list, replacing an entry with  $ID = ID_e$  if it exists. Note that the hops-value is incremented by one and the `Forward` flag is set to true.

At any time, the number of cars in the CarPark is the number of valid entries in the list. A valid entry is an entry whose `ReceiveTime` is at most `timeout` seconds in the past, where `timeout` should be bigger than the send interval. The `Forward` bit ensures that an information is only forwarded **once** after reception. This is necessary to accelerate the pruning process when a car leaves the CarPark, and also prevents poisoned reverse.

Some notes about the forward bit: We intended to extend the algorithm such that each car forwards the updates received from other cars. In order to reuse the same list of entries, but only forward the really new<sup>10</sup> update messages, we introduced the notion of this forward bit. It is set to true whenever a valid update is received for a car, i.e. the car is allowed to forward this information. It is reset to false when the update is sent. This simulates a (delayed) transmitter that accumulates messages to forward and sends them in a burst.

Without this bit, the car would forward information until it times out (and is removed from the list). As a consequence, the pruning process for cars that left would take much more time: Assume three cars  $C_A$ ,  $C_B$  and  $C_C$  parked in a line such that  $C_A$  and  $C_C$  cannot communicate with each other. Furthermore assume that  $C_B$  and  $C_C$  know that  $C_A$  is in the CarPark ( $C_C$  learned it via  $C_B$ ). When  $C_A$  leaves the CarPark (and no forward bit is used),  $C_B$  continues sending update messages including  $C_A$  until the information about  $C_A$  stored by  $C_B$  times out. Up to this timeout of  $C_B$ ,  $C_C$  received

<sup>7</sup>The time interval can be specified in the car configuration file.

<sup>8</sup>Note that the hops-value  $hops_e$  is incremented by one when stored, i.e. this means that messages with the *same* number of hops are accepted as new info.

<sup>9</sup>The timeout for 'in car park updates' can be specified in the car configuration file, it must be bigger than the send interval.

<sup>10</sup>New is an entry that was received since this car sent the last update message.

update messages about  $C_A$ , which it accepted. It now takes another *timeout* time until  $C_C$  forgets about  $C_A$  and does not forward this information anymore.

We can see two problems with the previous example: If we would have used the forward bit,  $C_B$  would only have forwarded once the presence of  $C_A$  and  $C_C$  would have timed out much earlier. Furthermore, we even get into big trouble if we do not use the forward bit: At the end,  $C_C$  sends update messages, but  $C_B$  has already timed out. This means, that  $C_B$  accepts those updates, sent by  $C_C$ , which leads to a kind of poisoned reverse:  $C_C$  will finally time out, but it is then  $C_B$  that believes that  $C_A$  is still in the CarPark, and sends this info to  $C_C$ , which believes it because it timed out. The forward bit in combination with the timeout solves this problem. If the timeout is bigger than the send interval time,  $C_B$  will time-out after  $C_C$  has forwarded the info about  $C_A$  and no problem arises. We usually used four times the send-interval time for the timeout.

### 6.2.3 High Availability of Paying CarParks

Paying CarParks contain computers that broadcast their availability to all subscribed<sup>1</sup> cars. During failures of such a computer, accurate information about the availability of the CarPark must be granted. This is solved by storing this information on all paying CarParks in parallel.

It is very hard to detect whether a CarPark is down. Therefore, all CarParks will always broadcast the information of all paying CarParks to subscribed cars. This requires either that all CarParks **always** have the most recent knowledge, which is very hard in case of message losses, a division of the network into two groups or others unpleasant corner cases.

Very similar to cars, each paying CarPark holds a list of all (paying) CarParks, containing the number of free places and a time indicating when this information was created. If a CarPark has some changes in its availability, it updates this list and broadcasts an update message containing the information of the list. Upon receipt of an update message of this kind, a CarPark updates its list if there is some more recent information available. This simple approach distributes the information among all CarParks and ensures that always the most recent information is taken into account. This very same update message is also received by (subscribed) cars, who extract the availability of all paying CarParks. To ensure nearly immediate updates for new cars entering the system, those update messages are sent periodically<sup>12</sup>.

This approach was simple to implement, because it allowed to reuse much of the modules used to send update messages by the cars. The format of the messages is the same, the only difference is the protocol number of the multicast update message.

**CarPark is Down - How to Obtain Entry/Exit Permission** If a CarPark  $CP_A$  is down, it cannot permit a car to enter or leave the CarPark, because the computer is down and therefore the information about the number of free parking lots is not available. In such a case, a neighbor CarPark  $CP_B$  is asked the permission to enter or leave. It is then this  $CP_B$  that modifies the availability of  $CP_A$  and sends an update message containing the new information.

The choice of the neighbor CarPark is of critical importance. All cars **must** choose the same CarPark. This is done by ordering the CarParks by distance to the destination CarPark and taking the closest that is *up*. If this is violated, two simultaneous entry requests by two cars to two *different* neighbor CarParks might be accepted even though there is only one free place. If we can assume that always the same neighbor CarPark is chosen, the only point of mutual exclusion to the number of

<sup>11</sup>The information is broadcast on the same multicast channel as all other multicast messages, but only received by subscribed cars.

<sup>12</sup>The time interval can be specified with the 'Update sendDelta' variable in the system configuration file s.xml.

free places in the original CarPark is this neighbor CarPark. If this is not the case, distributed mutual exclusion algorithms would have to be used to avoid the described situation. (Note that at least one CarPark must be up at any time to not forget the number of free places in CarParks.)

An (already mentioned) implementation issue is that the distribution of the parking lots is **always** handled by the CarPark itself and not by its neighbors. With other words: Each CarPark stores the number of free parking lots. This information is distributed among all paying CarParks. But each CarPark also stores which of the parking lots is used to facilitate the task of finding a free parking lot, once a car was accepted to park. This is mainly a list that indicates the status for each parking lot in the CarPark. This list is **not** distributed, it is only part of this simulation. When a car arrives and free parking lots are available (Note: for this information, this list does not need to be read!), the CarPark will assign a parking lot using this list, in our case coordinates where the car should park.

This design facilitates the implementation of the mutual exclusion that must hold on parking lots. If a CarPark is down, its neighbor CarPark only reduces the number of free parking lots, but does not assign a free location. This location must always be requested from the destination CarPark, even if its computer is down.

You might now ask yourself whether this is not a contradiction. Well, it is not, because this assigning of a location is only a simplification of the car looking itself for the free parking lot. In fact, this request does not have any meaning in reality, it is just a simplification for this simulation.

This explains why the `cp.exe` application must not be stopped during simulation. The simulation of a CarPark failure must not be done by killing the application `cp.exe`, but must also be simulated (right-click in the SimC on a CarPark and chose 'Shut down').

**Recovery of a CarPark** Upon recovery, a CarPark listens on the multicast channel for update messages for a certain time<sup>13</sup> and sets its counter of free parking lots. During this time, **NO** cars might enter or leave this CarPark. They have to wait until the CarPark has totally recovered. This restriction ensures that no simultaneous permissions are handed out by the recovered CarPark and its neighbor for the same parking lot.

This might sound difficult to implement, but it is not. A CarPark that is down, replies to (nearly) all requests with 'Down'. This is kind of cheating, but allows the car to learn fast that a CarPark will not handle its request. The car then switches to the next closest neighbor CarPark until it finds one that is up. During recovery, the CarPark does not reply at all until it recovered and then sends the answer to the request.

#### 6.2.4 Finding the Fastest Route to a Destination

Finding a route to a destination has two components:

- Find the closest CarPark that has available parking lots
- Find the best route to this CarPark.

Finding the closest CarPark is simple if the Euclidean distance to the centers of the CarParks is taken. For this implementation, also the decision to choose the CarPark is defined as simple as possible: The closest CarPark that has available places is taken.

---

<sup>13</sup>This time is called the `recoveryTime` and can be specified in the system configuration file `s.xml`. It must be bigger than the `Update sendDelta`, the time interval in which updates are sent automatically.

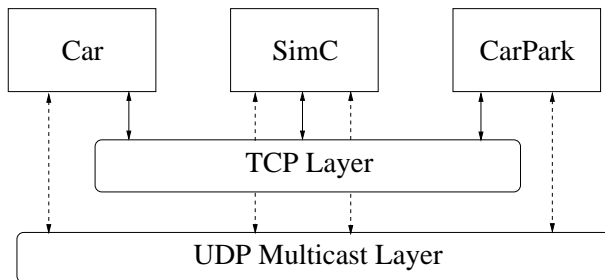


Figure 15: Major components of the simulation and their interaction with the network layer.

In reality, this might not be the optimal choice. The price of the different CarParks, the probability to still obtain a place when arrived at the CarPark and the paths to walk from the CarPark to the final destination should be taken into account.

Once the CarPark is known, the best, in our case the fastest route to the CarPark is calculated. Streets can have different speed limits, so do cars<sup>14</sup>. This routing is done by the map utility and uses dynamic programming.

### 6.3 Design Overview

Figure 15 shows the major components of the simulation and their interaction with the network layer. The UDP multicast network layer simulates a WLAN or multipoint Bluetooth. The signal strength of those emitters cannot be simulated with UDP multicast - all listeners will receive each message according to a certain probability (depending on the network stress). To simulate a limited signal strength for messages sent by a car, cars will only accept those messages if the sender is within a certain distance.

The different messages that are sent on the network are the following:

- ▷ *Free update*: Update message that contains information about free CarParks (UDP).
- ▷ *Paying update*: Update message that contains information about paying CarParks. Those messages can be received by the cars independently from their location, we assume that the paying CarParks have emitters to broadcast over the entire geographical environment (UDP).
- ▷ *Time synchronization*: Message that contains time information. This message is used by the cars for synchronizing themselves (UDP).
- ▷ *Request/Grant*: Message that are used for entering or leaving a CarPark (TCP) and for granting access to a this CarPark.
- ▷ *Signup*: Message used to register the car and the CarPark to the simulation center (TCP).

The Car, CarPark and SimC are standalone entities in the sense that each of them are running in a separate process. It is hence possible to run them on different computers.

<sup>14</sup>The GUI for the Car allows to change the speed of the car. This allows to change the speed without any restrictions by streets or the car.

## 6.4 Final Remarks

The current implementation of the prototype is running on Windows XP with the .NET framework version 1.14322. The user needs also cygwin installed in order to compile the different executables. The current 0.1 version of the prototype is available at

`http://lpdwww/sbaehni/work/carPark/carPark.html`

For a more complete description of the implementation please see the Web site.

## References

- [1] Centre of Excellence in Information and Communication Technologies (CETIC). Applied Research Laboratory. See <http://www.cetic.be>.
- [2] Community panel. Available at <http://renoir.info.ucl.ac.be/twiki/bin/view/INGI/CommunityPanel>, 2004.
- [3] Distributed car park. Available at <http://lpdwww.epfl.ch/sbaehni/work/carPark/carPark.html>, 2004.
- [4] S. Baehni, P. Th. Eugster, and R. Guerraoui. Data-aware multicast. In *In Proceedings of the 5th IEEE International Conference on Dependable Systems and Networks (DSN 2004)*, pages 233–242, June 2004.
- [5] M. Brahami, P. Th. Eugster, R. Guerraoui, and S. B. Handurukande. Bgp-based clustering for scalable and reliable gossip broadcast. In *LNCS volume of the post-proceedings of the Global Computing 2004 workshop*, 2004.
- [6] Bruno Carton and Valentin Mesaros. Improving the scalability of logarithmic-degree DHT-based peer-to-peer networks. In *Euro-Par 2004, Pisa, Italy*, August 2004.
- [7] Denys Duchier. The MOZart Global User Library, 2003. Available at <http://www.mozart-oz.org/mogul/>.
- [8] P. Th. Eugster, R. Guerraoui, and P. Kouznetsov. Delta-reliability: A probabilistic measure of broadcast reliability. In *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS 2004)*, March 2004.
- [9] Erik Klintskog, Zacharias El Banna, Per Brand, and Seif Haridi. The DSS, a middleware library for efficient and transparent distribution of language entities. In *Distributed Object and Component-based Software Systems Minitrack in the Software Technology Track of the 37th Hawaii International Conference on System Sciences (HICSS-37)*, Big Island, Hawaii, USA, January 2004. IEEE Computer Society Press.
- [10] Per Brand Luc Onana Alima, Sameh El-Ansary and Seif Haridi. Dks (n, k, f): A family of low communication, scalable and fault-tolerant infrastructures for p2p applications. In *3rd IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2003), 12-15 May 2003, Tokyo, Japan*. IEEE Computer Society, 2003.
- [11] Mozart Consortium. The Mozart Programming System, version 1.3.0, 2003. Available at <http://www.mozart-oz.org/>.
- [12] L. Onana, A. Ghodsi, P. Brand, and S. Haridi. Self-correcting broadcast in distributed hash tables. In *OPODIS 2003: 7<sup>th</sup> International Conference on Principles of Distributed Systems*, 2004.
- [13] Luc Onana, Sameh El-Ansary, Per Brand, and Seif Haridi. DKS: A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications. In *CCGRID*, May 2003.
- [14] UCL and CETIC. P2PS: Peer-to-Peer System Library. Available at [http://www.mozart-oz.org/mogul/info/cetic\\_ucl/p2ps.html](http://www.mozart-oz.org/mogul/info/cetic_ucl/p2ps.html), 2004.