DISS. ETH NO. 17611

Hosting distributed software projects: concepts, framework and the Origo experience

A dissertation submitted to
ETH ZURICH

for the degree of
Doctor of Technical Sciences

presented by

TILL GASTON BALZ BAY

Dipl. Informatik-Ing.ETH, ETH Zurich

born June 13th, 1978

citizen of
Bern BG, Bern

accepted on the recommendation of
Prof. Dr. Bertrand Meyer, examiner
Prof. Brian Fitzgerald, co-examiner
Prof. Roger Wattenhofer, co-examiner

2008

Acknowledgments

I would like to use this opportunity to thank the people that helped me during my thesis.

First of all I want to thank Professor Dr. Bertrand Meyer for hosting me in his group, the Chair of Software Engineering at ETH Zurich. Special thanks also go to the co-examiners of this thesis: Prof. Brian Fitzgerald and Prof. Roger Wattenhofer.

I thank my friend and office-mate Michela Pedroni for being the best companion and reviewer there is. In the same way Claudia Günthard, Dr. Manuel Oriol, Bernd Schoeller and everyone from the group receive my thanks.

I would like to thank all students who collaborated with me: Patrick Ruckstuhl, Peter Wyss, Beat Strasser, Dominik Schneider, Marco Zietzling, Dennis Rietmann, Julian Tschannen, Rafael Bischof, Martin Seiler, Urs Doenni, Benno Baumgartner to mention the most important ones.

Finally my biggest thanks go to my friends and family – I would have never been able to achieve this without your love, support and the great time we are having together!

Abstract

Developing software systems is a complex activity that involves numerous tasks critical for a successful release. The increasing size of systems, software maintenance, versioning and distributed global development complicate the development; a software development platform that integrates into the development process can automate and simplify considerable parts of that process.

Just as important as the technical activities of software development, are management and communication tasks: recording project events, managing project Wikis and web pages, sending out notifications, reconciling changes, and many others. These tasks become ever more delicate with the increasingly distributed nature of modern software projects, small as well as large. If not handled properly they can not only consume considerable time but also, just like bugs and other flaws in technical tasks, cause considerable damage.

Origo is a comprehensive platform for addressing project needs by providing such facilities as project web pages (both editable and generated), forums, bug tracking etc. All the facilities are also available through a program interface (API), allowing development tools and environments to invoke Origo mechanisms automatically upon completion of such specified project events. An event can be the publication of a software release, a commit into the configuration management system, a modified Wiki page, a posted blog or a comment left on the project site; environments for which a specific Origo plug-in already exists include VisualStudio, Eclipse and EiffelStudio. Internally, Origo relies on a peer-to-peer middleware architecture supporting the integration of such application components as web and Wiki servers, database servers, business logic, configuration management, identification, access control, load balancing.

Hosted by ETH Zurich, Origo is free to any project, open-source or closed-source, and designed for scalability. Origo Core is the underlying P2P based application construction framework that is used to building Origo. The framework is general and can satisfy architectural requirements that go beyond the construction of a platform for hosting distributed software development projects.

Zusammenfassung

Software Entwicklung ist eine komplexe Aktivität. Zahlreiche Tätigkeiten tragen zu einer erfolgreichen Veröffentlichung einer Applikation bei. Die wachsende Grösse von Systemen, deren Unterhalt, die Versionnierung sowie die global Verteilte Entwicklung machen Software Entwicklung immer komplexer. Eine in den Entwcklungs Prozess integrierte Plattform kann beträchtliche Teile davon automatisieren oder vereinfachen.

Ebenso wichtig wie die technischen Tätigkeiten, sind Management Aufgaben und Kommunikation in einem Projekt. Projekt Web Sites müssen upgedated und unterhalten werden, Ankündigungen werden versandt, Veränderungen aufgenommen und vieles andere. Sowohl fuer kleine, wie auch für grosse Projekte werden diese Aufgaben schwieriger, je verteilter das Projekt Team arbeitet. Werden diese Tätigkeiten nicht sachgemäss ausgeführt kann das zu Verspätungen, aber auch zu Fehlerhafter Funktion führen mit beeinträchtigenden Folgen führen.

Origo ist eine umfassende Softwareentwicklungs Platform, die die Bedürfnisse moderner Entwicklungs Teams abdeckt. Jedes Projekt hat generierte und editierbare Webseiten, Diskussionsforen, Issue Tracking etc. Die gesamte Funktionnalität der Platform kann durch eine Programmierschnittstelle (API) angesprochen werden. Neben der Schnittstelle existieren plug-ins für die gängigen Entwicklungs Umgebungen.

Der Kern von Origo basiert auf einer P2P Bibliothek, die die Integration von Externen Applikationen auf einfache Art ermöglicht. Diese Infrastruktur ist erweiterbar, erlaubt ein Update zur Laufzeit und unterstützt die Verteilung aufkommender Last.

Die ETH Zürich hosted die Origo Plattform; jedermann kann sowohl open-source wie auch closed-source Projekte damit veröffentlichen. Die verwendete Bibliothek Origo Core kann zur Konstruktion beliebiger verteilter Anwendungen und Plattformen verwendet werden.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Origo: An overview

The open-source development community has built significant software over the last years. The best example is the ongoing evolution and success of Linux and its variants. The community manages complex, shared and globally distributed development efforts using a volunteer development staff with limited institutional memory (developers come and go as the projects continue to evolve). Most software developed in universities follows a similar pattern.

Key to the success of many open-source projects have been the adoption of *code stewardship* principles and the use of distributed software development platforms such as SourceForge [1]. For large-scale, shared development to be successful, especially in an environment where the development staff have other pressing duties, the development, testing, packaging, distribution functions, and user-community management must be supported in a integrated and coordinated way.

In this work, we describe Origo, a software development platform for integrating code stewardship technologies which takes distributed development platforms to the next level and enables them to be extended in the future. The platform can serve the needs of scientific application development, maintenance, distribution, and user collaboration as well as the needs of the open- and closed-source development communities.

Stewardship of software and applications is complex. Use of extant tools and services for software stewardship and integrating them into a workflow takes time away from developers and does not constitute the core business. This complexity is due in part to the sheer number of services available, the rapid emergence of new tools, the expertise required for integration of multiple tools into a single development environment, as well as the heterogeneity and independence across services.

Origo (Latin for "The Source") combines several applications and tools to allow developers to collaborate over a network. The combination of these applications forms the information systems used in software projects. "Releasing software early and often", one of the mottoes of modern software development, requires that such an information system fits well into the development process. Often, development teams are the first adopters of emerging tools and technologies that support and accelerate collaborative work. As a consequence a software development information system has to be able to integrate new applications when they become important for a process.

The Origo platform provides a generic replacement for any ad-hoc combination of applications and improves the state of the art of existing development platforms. Released in August 2007, the platform already hosts by December 17th, 2007 more than 628 projects with over 1566 developers worldwide.

The novel services Origo offers are an application programming interface (API), an innovative display of events of project life and the possibility to host open- and closed-source project development.

Offering an API allows integration of Origo into any development process (see Section 1.1).

Origo relies on a network layer programmed using the JXTA [2] peer-to-peer protocol. This makes it possible to add new applications by extending them with JXTA libraries to handle the transport layer.

1.1 Using Origo

Origo (http://origo.ethz.ch) is an open-source information management platform for software projects. The platform enables a team of developers to track their own projects and those of other teams. For a developer, Origo encourages collaboration with several development teams and allows working both on open and closed-source projects. Every Origo project has a web page that can be reached over a sub-domain (http://yourproject.origo.ethz.ch).

The platform does not impose any particular development model, technology or tool; the development proper happens outside Origo. The following paragraphs describe what Origo offers both for development teams and for users of the software projects that are hosted on Origo. For a complete discussion how to use Origo, refer to Chapter 6.

1.1.1 Projects and people

The software development platform manages projects, their development teams and their user communities. Once registered as an Origo user, one can hold any of three roles for a given project: *project user*, *project member*, *project owner*. These correspond to different interests:

- *members* and *owners* are part of a project's development team. They both can modify the wiki pages of the project web site, create releases on the download area and commit code into the Subversion repository. They can also post blogs, report issues and generally modify all content on the project pages. Only *owners* may add new developers to a project team.
- *users* can only report issues, write forum posts and comment content on project pages.

The low number of different roles people can hold keeps the rights management for all actions concerning the project pages simple. This simplicity also contributes to the usability of Origo: users and developers do not need to read documentation to start using the platform.

1.1.2 Basic features

The basic features of an Origo-hosted project are the usual services on which development teams rely today: a configuration management server for hosting the code; documentation and communication possibilities; ways to report and manage issues; a place to publish project releases. More advanced features are detailed in Section 1.2.5. Every Origo project has the following features:

- Public and private wiki pages with WikiMedia syntax.
- Subversion repository with web user interface.
- Issue tracking with public and private issues.
- Blog, forums, comments, tags and screenshots.
- Release download area with mirroring.

The project web page, of which an actual example appears in Figure 1.1, has navigation links that lead to the home page, download section, screenshots, documentation, the forum page, the blog, the issue tracker and the

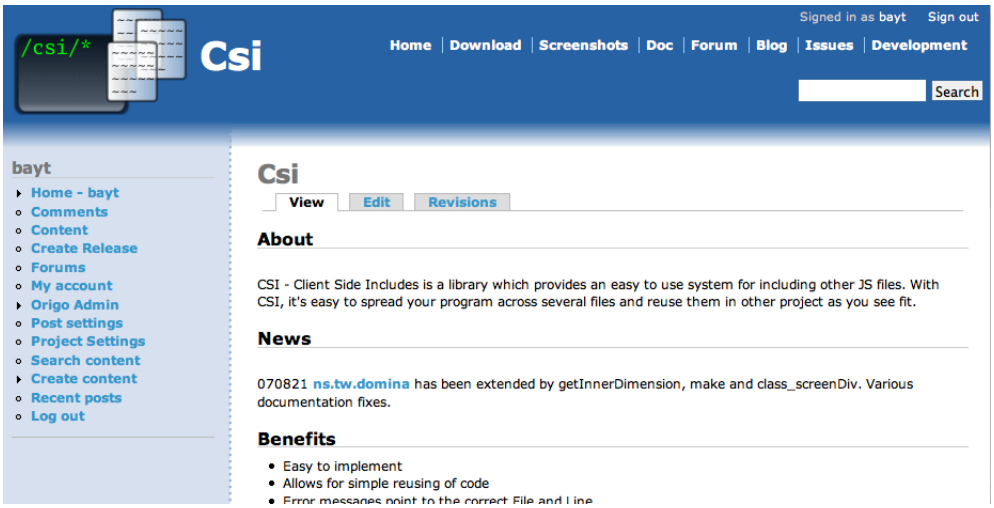


Figure 1.1: Typical Origo project page - http://csi.origo.ethz.ch

development page. A project includes two kinds of pages: editable and generated. Editable pages which project members can freely create and update, use the wiki format; they include the home page, the screenshots, the documentation and the development pages. Generated pages are the download area (which lists all project releases), the forum, the blog and the issue tracker.

Fundamental functionalities, repeated on every page (currently in the left-side menu, see Figure 1.1) include: the user and project settings, creation of new content, the request form for project creation and pages that allow tracking changes.

Also on every page are the search functions of several kinds. First, all projects hosted on Origo can be searched with Google Custom Search [3]. Every project page can also be searched separately. One can search for Origo users. The fourth search retrieves tagged issues. The searches are implemented using Generic Component Lookup (GCL) [4]. GCL is a search system that identifies different dimensions in the data to search separately, and weighs the results for each dimension. Sorting results according to an arithmetic combination of the weights fine-tunes and improves them. This way users can both search all the tags and the text of a reported issue separately but then receive the results combined.

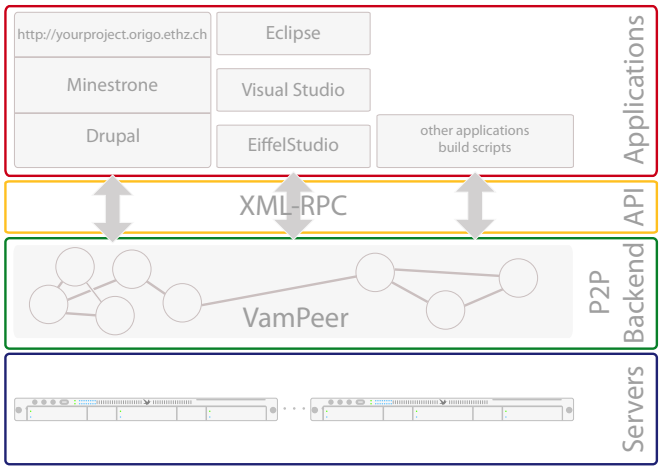


Figure 1.2: Origo architecture

1.2 Architecture

The key design goals for Origo were scalability, extendibility and language independence. To achieve these aims, Origo uses a peer-to-peer (P2P) back-end relying on widely supported communication protocols. Figure 1.2 shows the architecture of the platform.

Origo is running on multiple servers and is built following the model view controller (MVC) pattern. In Figure 1.2, the P2P back-end represents the controller and the applications represent the different views that are available. One of those views is the project page of a given Origo project; another would be the work items that are displayed in the Eclipse plug-in for Origo[1]. The views and the controller communicate using XML-RPC. For communication inside the controller, Origo uses the P2P framework VamPeer [5], itself based on JXTA [2]. Some nodes of the back-end provide access to databases that are themselves representing the model in the MVC pattern. For a complete discussion of the architecture, the communication infrastructure as well as the search mechanisms, refer to Chapters 3, 4 and 5.

1.2.1 Back-end

The back-end forming the controller of Origo is built using different peer-to-peer nodes. Besides acting as controller for the platform, the nodes of the back-end also provide access to several collaborating services that are used in

[1] http://origo.ethz.ch/download

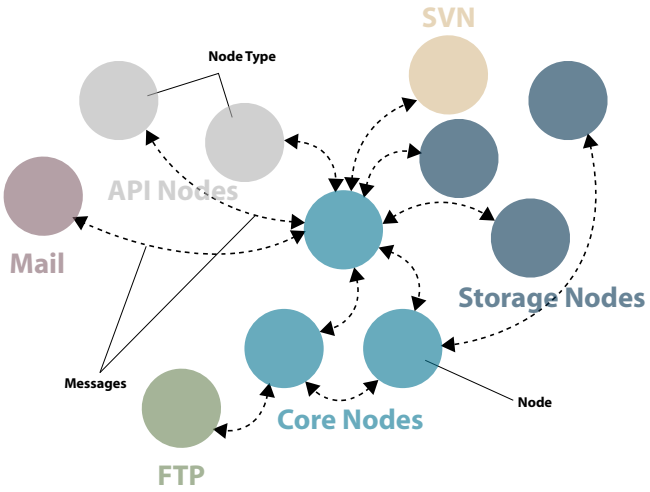


Figure 1.3: Nodes and node types in Origo

Origo. Services include the database server, the Subversion servers, the FTP and the mail server; all of them exist as nodes of the back-end. The JXTA protocol is the infrastructure that they use to communicate. The back-end relies on the following notions that we will detail below: *nodes* and *node types*, *messages* and *use cases*, *core nodes* and *API nodes*. After describing these, a short discussion of the choice of P2P framework follows.

Nodes and node Types

Each *node* represents a service used in the back-end (see Figure 1.3). A *node type* regroups a set of *nodes* that all provide the same service. Each *node type* has its own policies for *message* processing. Each *node* of a given *node type* is equal and any incoming *message* for a *node* can be treated by any other *node* of the same type.

Messages and use cases

The core nodes contain the code representing the *use cases*. A *use case* is the description of how the Origo back-end reacts to an incoming XML-RPC API call. After the API nodes receive a call, *messages* are sent to the core nodes that then send *messages* to all *nodes* taking part and performing some action in that particular *use case*.

Core nodes

The core nodes are the controller of Origo and contain the code of the *use cases*. They manage all interactions between other nodes. Load balancing for all *node types* is implemented in the core nodes. The core nodes are aware of all *node types* existing and know how to use them to perform each of the *use cases*.

API nodes

The API nodes are the interface to access the Origo back-end. API nodes listen to incoming XML-RPC calls as a daemon on a port on the server machines of the platform. The API nodes expose parts of the *use cases* that are encoded in the core nodes. They are not load balanced by the core nodes. For load balancing incoming XML-RPC calls, traditional web server load balancing techniques like round robin IP address resolution, or LVS systems should be used.

A list of the API calls currently offered can be found online[2]. The calls offered allow access to all information stored in an Origo project. There are calls to enumerate the work items for a project, calls to upload and publish project releases, calls for issue management as well as general calls handling access and rights management, login processing and management of user information.

Internal API nodes

To avoid possible denial of service attacks to API nodes, parts of the exposed API have to be hidden to outside applications. The API call for creating a new Origo user is an example for a call that has to be hidden, because bots would be able to create uncontrollable numbers of Origo users if it were available. Hidden calls are only available to certain views - in the case of the user-creation call, the view that has access to it is the Origo project web page (in Figure 1.2 the web page of an Origo project is the stack Drupal / Minestrone / http://yourproject.origo.ethz.ch in the applications layer).

Storage nodes

The storage nodes maintain connections to database servers. All data used by Origo is stored in the databases managed by them. The storage nodes handle all communication and authentication with the databases. The databases

[2] http://origo.ethz.ch/wiki/origo_api

store both data on users (user ID, name, password hash, email address and an application key) and on projects (name, description and a logo) as well as the association of users and projects, associations between users (development teams), access policies for all resources managed by the platform, role management information, and session management.

Custom nodes

An important part of Origo are the services used for software development. The back-end contains three custom nodes that wrap these services: One node for configuration management (at the moment this is done with Subversion), one node for file upload when creating software releases (implemented with an FTP server) and finally a node that can send mail. Figure 1.3 shows the three nodes together with the other node of the back-end.

These custom nodes wrap an external server application as a node for the back-end. These server applications all have their own user management mechanisms and access protocols. The custom nodes of the back-end configure these server applications to integrate them fully into Origo. The login credentials for these services are created by the custom nodes that can write configuration files and execute processes. This allows to integrate any application into Origo. The way developers are working evolves and the configuration management servers that are used today might be replaced by new servers that have different functionality in the future; using custom nodes allows such evolution and adaption of the back-end.

1.2.2 Scalability

Nodes of the same type perform the same service within Origo. Nodes receive messages and then return the results to the node that sent the message. They are not aware of the other nodes of the same type within their group of nodes. Nodes have unique identifiers inside the peer-to-peer back-end and can be addressed using this identifier. How to balance load among nodes of a same type can vary and it is left to the implementer of the node type. As a simple solution, round robin load balancing is used for the core and storage nodes.

1.2.3 Extendibility

Two attributes of Origo allow its adaption. The first capability lies in the nature of the nodes. Nodes do not need to be on one single computer; they can be distributed across a number of machines. Nodes of the same type

can be on different machines. By measuring load and performance on the running platform, a favorable distribution across machines can be found.

The second attribute making it possible for Origo to react to change is that nodes can join the back-end at any time. If at one point in time the number of nodes of a certain type is not sufficient anymore (this can be caused by increased load for example) - more nodes of that type can be started. These nodes will then register themselves within the back-end and start processing messages. The inverse case of nodes leaving a system can also happen and can be used to react to changing needs. This mechanism is also used to update the running platform dynamically whenever new or updated nodes are becoming available.

1.2.4 Language independence

The Origo back-end uses the JXTA [2] peer-to-peer protocol and benefits from the multiple implementations of JXTA that make it language independent. Both the communication inside the back-end as well as the interface to the outside are language independent. The messages exchanged are simple key value pairs of strings that are sent from one node to another. The calls Origo can receive from the outside world are similar and reach the system using the XML-RPC [6] transport protocol.

Chioce of P2P Framework

The available peer-to-peer frameworks today include Chimera (was Tapestry),[3] Pastry [7], Chord[4] (distributed hash functions and the Self-certifying File System [5]), GNUNet[6], XNap[7], and the Peer-to-Peer Trusted Library[8]. Some of these systems address the necessary P2P networking requirements sufficiently, others provide routing algorithms that are adapted to a specific peer-to-peer application (like for example file sharing). None of these frameworks, however provide an application construction framework. A peer-to-peer application construction framework is general and abstract enough to support building P2P applications that go beyond file sharing and instant messaging. The existing application construction frameworks are Juxtapose (JXTA) [2], Jini [8], and OogP2P[9]. Jini is implemented only

[3] http://current.cs.ucsb.edu/projects/chimera
[4] http://pdos.csail.mit.edu/chord
[5] http://www.fs.net/sfswww
[6] http://www.ovmj.org/GNUnet
[7] http://xnap.sourceforge.net
[8] http://sourceforge.net/projects/ptptl
[9] http://www.duke.edu/~cmz/p2p

for the Java language and OogP2P does not provide sufficient functionality to be used in Origo. This justifies the use of JXTA for implementing the back-end.

JXTA has extensive functionality. The JXTA specification contains protocols for routing, message passing, discovery of peers and support for secure communication using HTTPS. In addition, multiple language bindings (for Eiffel, Java and C) exist. JXTA is open-source, widely used, well supported, easy to extend and is general and abstract enough to support the functionality required by Origo. With JXTA, Origo propagates the messages. JXTA implements all routing and other communication protocols for such exchanges in Origo in a technology-independent fashion.

1.2.5 Novel user features

Together with the scalable, extendible and language independent design of the platform (see Section 1.2), Origo innovates through the following features:

- Besides the user interface already sketched, Origo provides a programming interface (API) enabling application developers to hook their processes and tools programmatically to the platform.
- Origo gives developers and users of a project a concise overview of the state of the projects.
- Origo allows hosting both open and closed-source projects.

We are now describing each of these three innovations in more detail:

Integration into the development process

The Origo API enables integration of the platform into any development process. It is implemented using XML-RPC [6]. XML-RPC is a simple, open-source specification and implementation for remote procedure calls between disparate and heterogeneous software systems. It uses HTTP/S as transport protocol and XML for message encoding. Messages are method calls with their argument data. Currently a wide range of languages provide XML-RPC libraries, including Eiffel, Java, C, Python, Perl, Objective-C, PHP. Since XML-RPC uses HTTP/S as transport protocol, it is easy to implement if no existing language binding can be used.

Every software project creates deliveries of its code. In some cases the delivery is an application that users can download, in other cases the delivery consists of a library that can be reused. All languages, operating systems and

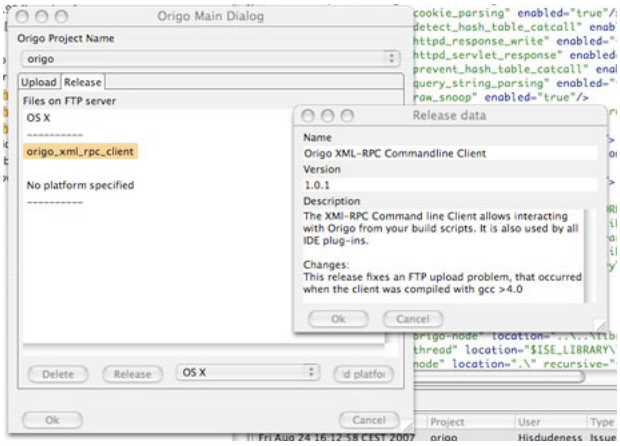


Figure 1.4: Release creation dialog in the Origo Eclipse plug-in

tools rely on the regularly recurring activity of building a delivery, driven by scripts. In spite of the wide diversity of tools and processes, the delivery step follows a common pattern. It involves a number of actions scheduled not manually but through a script like a *Makefile*, an *ANT-Script* or a *Visual Studio Solution File*. Once the script has been run, the next step is to publish the delivery online. Publishing a delivery can with the Origo API be integrated into the scripts. One of those scripts simply has to use the API call to publish a release for a project on its Origo page; that last step is also automated. There are many other API calls available for a project (see the complete list online[10]). This example illustrates how the Origo API enables hooking into a development process.

IDE Integration: Similarly to build script integration, the platform can be used directly from integrated development environments. Currently, plug-ins for EiffelStudio, Eclipse and VisualStudio exist.

Figure 1.4 shows the Origo Eclipse plug-in. Using the plug-in for an IDE, users can choose files to upload to the project page, specifying release and platform information. All three IDE plug-ins offer the same functionality and the release creation dialog looks the same for all three IDE plug-ins. They all use the XML-RPC API to send calls from the IDE to Origo.

Besides integrating release creation, the IDE plug-ins also offer access to the work item overview that is discussed in the next section.

[10] http://origo.ethz.ch/wiki/origo_api

Work item overview

A member of a collaborative project needs to know what other team members are doing. The most important resources are the source code files; projects will typically rely on configuration management such as Subversion to enable concurrent access by multiple developers. Apart from managing the code with such a tool, developers typically also like to be informed of changes to the source code in other ways. An important functionality of such systems is the ability to send mail to project members on occurrence of specific events such as commits.

Origo generalizes this concept by enabling projects to hook up various actions to many possible events of project life. In Origo, all resources that can experience modifications are tracked and the changes summarized in the work item overview of a project page (see Figure 1.5).

The five resources tracked by Origo for each project are changes to source files, issues that are reported or modified, wiki pages that are edited, blogs that are posted and releases that are published. Figure 1.5 shows the work item overview page. In the Figure, the tab for the Origo project itself is active and it shows the work items of the project at that point in time. Each listed work item links to a corresponding page that shows the changed resource. When a new resource is created (for example a new blog posted) - the link points to that resource. If an existing resource is modified, the link points to a page showing the differences between the old and the new version of the resource (for example when a typo on an existing wiki page is corrected). The Figure shows also other tabs for other projects. The tabs on the left are all the projects of which the user (here: bayt) is a member. After the projects that a user belongs to, the tabs for bookmarked projects are enumerated. Every project on Origo can be bookmarked by a user and then the work items for that project are shown on the overview page accordingly. Whenever new work items for a project are published, its corresponding tab is highlighted.

For each project showing on the work item overview page one can configure both the mode of notification (receiving a mail for an update on a resource or Listing the updated work item on the overview page) and it can be set which of the five work item types should be tracked. This allows setting the level of information a user or developer desires individually for every project. For all work items and projects an RSS feed is generated and can be used to get information on work item updates as well.

As mentioned above, also the IDE plug-ins show the work item overview as part of their user interface. Just like the work item overview page, the IDE plug-ins link the displayed work items back to Origo, or show the updated

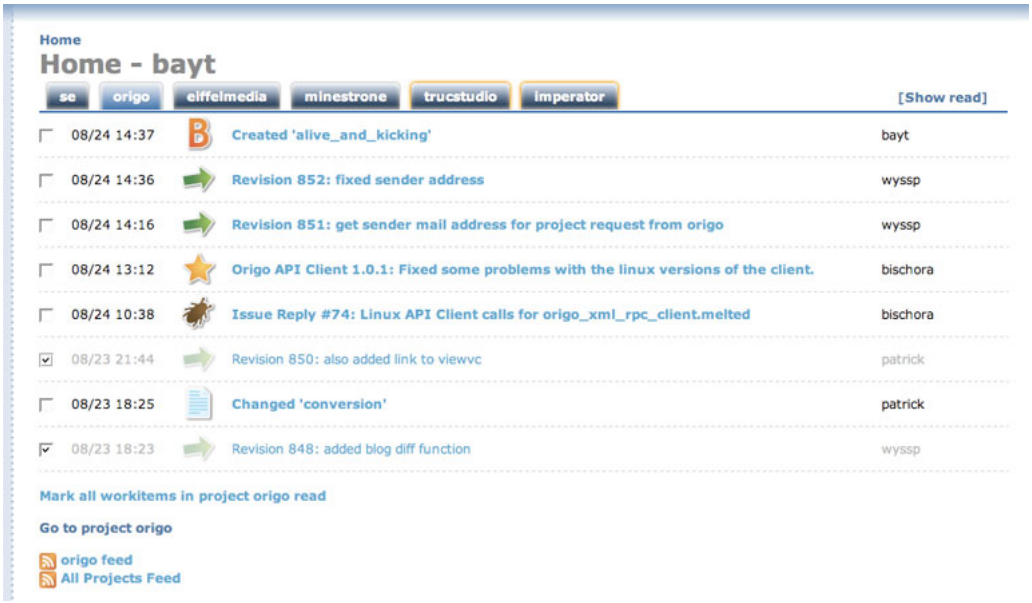


Figure 1.5: Work item overview page for multiple projects

differences directly inside the IDE. For screenshots of the IDE plug-ins see the screenshot page online[11].

Open- and closed-source projects

The third contribution of Origo is the hosting of closed-source projects. Origo helps its users produce and maintain better software, faster, more effectively. It is not its role to push a political agenda for either open- or closed-source development. There are all kinds of reasons for a project to prefer a closed-source model. This is not only true in the software industry, but even in a purely academic environment where some work on projects needs to be kept closed until time for publication is suitable. The open or closed-source nature of an Origo project determines the visibility of its work items for its non-members; source code commits for example can only be seen and accessed by developers that are members of a closed-source project.

1.3 Related Work

This section presents other existing software development platforms as well as other service integrating middleware architectures.

[11] http://www.origo.ethz.ch/wiki/screenshots

1.3.1 Development platforms

Software development platforms are not new; the most popular platform known today is Sourceforge [1] with 155'000 projects and 1.5 million registered users. Others are discussed in Chapter 2.

The following points are where we feel that Origo innovates in comparison to other platforms:

- Origo is open-source.
- Origo has an API.
- Origo hosts open- and closed-source projects.

1.3.2 Middleware architectures

An important part of Origo is its architecture that separates the external services needed in Origo from the back-end of the platform. This has two effects: (1) it allows to scale up the platform if needed, (2) it allows the integration of different implementations of a given service.

Middleware for dynamic adaptation flourished in the past years. As an example, Linda-like [9] coordination media such as AOS [10] use a shared tuple space to decouple applications. Origo is by nature more flexible as it takes into account replicated nodes.

Contrary to systems like HydroJ [11], LuckyJ [12], Fractal [13] or Service Groups [14] where the final recipients of messages are chosen by the infrastructure, Origo defines the entry points but lets the node types define their own policies to treat messages. Origo is similar to Matrix [15] in that it specializes in providing an infrastructure to build distributed applications of a particular type. In the case of Matrix however, the applications that are built have precise requirements in terms of distribution (load balancing of the servers during peaks) while in the case of Origo the main requirement is extendibility. In the end, Origo is much more expressive than Matrix (that could be coded using Origo nodes).

Origo nodes integrate already existing applications and make them communicate independent of the language in which they were programmed; this is a very common characteristics for a middleware infrastructure like CORBA [16] or web services [17, 18]...). Comparing to other infrastructures, Origo nodes include the necessary tools to integrate business applications such as identity management and information controller components. They also rely on a language independent communication layer (XML-RPC

provides the external API, JXTA enables internal communication). This enables us to focus only on the parts of the system that really matter: the application logic and applications to compose using Origo nodes.

Chapter 2

The need for better support for distributed development

A platform for distributed development consists of several specialized services. These services are used to manage the distribution of source code (configuration management systems), to support software and project documentation, to enable user feedback and communication (web sites, forums and issue reporting), to publish software releases and also to facilitate project management for the development team.

Each service can be implemented with multiple tools. The choice of a tool to implement such a service, depends on the development process of each particular team and on the availability of tools. Choosing a configuration management server, for example, depends on the development process of a team: centralized configuration management like CVS or SVN vs. distributed configuration management like GIT or Bazaar. And the selection of tools for that purpose depends on their features as well – after years of using CVS, a team might want to switch to SVN because it is an improvement in functionality.

Platforms for distributed software development are always a heterogeneous conglomerate of specialized services targeted at one of the activities in the development process. The challenge when building a platform is to unify the services in a consistent and developer-friendly way while not compromising the specialized service as such. This unification of services has to remain easy to extend with other, new services that might become desirable in a development platform.

The following sections describe the argument for development platforms as such, look at frameworks for service and application composition and identify their challenges. Then we examine how existing distributed software development platforms address those challenges; we discuss the needs of de-

velopment teams at universities and finally show how Origo improves the state of the art both in its architecture as well as with its functionality.

2.1 Ad-hoc composition versus platform

Each of the tools in a platform is used to fulfill one of the common purposes within a development process: code management, documentation, communication, etc. The tools evolve, become better or are also replaced with new ones. Development teams are early adopters of emerging technologies and will not want to continue working with outdated tools for which better alternatives exist. There are two ways of setting up a work environment for a development team. Ad-hoc setup of tools versus integrating tools to become a platform.

2.1.1 Arguments for ad-hoc composition of development tools

Often the ad-hoc composition of a popular selection of tools builds the software development platform that teams choose to satisfy their needs.

Many existing tools are now aware of others that are used for development; an example for this would be a bug-tracking system that is able to extract references to entries in its bug database from comments from source code commit messages - another example is a configuration management system that already delivers with and example scripts showing how to hook it up with a mailing-list server. As developers are early adopters of new and useful tools, they can simply select a new tool to use in their workflow as it becomes available. Popularity of a tool often has an effect on its integration possibilities with other popular development tools.

2.1.2 The case for an integrated platform approach

There are a number of arguments as to why a software development platform integrating different tools is preferable over this ad-hoc composition of tools.

Even on that small scale, if integration and interaction of used tools becomes more available and popular, the ad-hoc scenario fails to address two important needs in distributed software development: self documentation and indexing.

The development process should document itself. The small scale interaction and integration that exist for some tools used for software development are a step in the right direction, but the activity of developing software is

sufficiently complex and time consuming that the tools should really establish a concise track of all activities happening in project life. Such automatic self documentation can only be achieved when the different tools are used within an information system that orchestrates their interactions and keeps track of all generated data within the process.

As development progresses and data is accumulated the need for good search capabilities grows; again it is the integrating development platform that can provide a unified interface to search across all different data that is part a projects life. These two arguments for integrating tools used in development establish the need for building information systems to support development.

2.2 Challenges of frameworks for service and application composition

Frameworks for application composition have to solve a number of problems. A framework must enable the exchange of data among the different applications that are part of it; it has to provide a way to describe use-cases that define how the applications that are glued together with the framework should interact. Application communication involves protocols and technologies which must both have to be sufficiently abstract to give room for heterogeneity. In large scale application composition, a degree of distribution has to be made possible by the application composition framework; and management-related requirements like dynamic update and extension, logging, reporting and monitoring capabilities have to be addressed as well.

Middleware frameworks that address these requirements have existed for some time now. Frameworks like CORBA [16] or web services [17, 18] allow composition of applications to bigger information systems in a technology-independent way. Dynamic updating of middleware can be achieved with Linda-like [9] coordination media such as AOS [10].

For Origo, we wanted to address requirements using a peer-to-peer based middleware. Origo takes advantage of the dynamic updating functionality as well as the distribution in a P2P network of nodes. The architecture of the back-end is described in Chapter 3.

2.3 State-of-the-art software development platforms

Platforms for distributed software development and for publication of the releases have existed for some time. In this section we discuss the most important ones and compare them to Origo.

SourceForge.net-like

The most popular platform is SourceForge, whose success is attested by 155'000 projects for 1.5 million users in September 2007. The SourceForge story illustrates the limitations of first-generation integrated platform solutions. Around 2003, it became clear that developers were migrating en masse, for configuration management, from the venerable CVS to the newer Subversion (SVN) system. It took SourceForge 18 months from the announcement of SVN support to its actual availability. This can be ascribed to an architecture that fails to reconcile the need for continuous adaptation with the requirement of round-the-clock availability of hosted projects, causing SourceForge in this case to become a victim of its own success.

The other issue with SourceForge is that it is very hard to integrate it into a development process. The platform offers no API and it is not really offering a very convenient web interface either. When releasing new software a developer has to upload the files belonging to a release to an anonymous ftp server account. Then he has to log on to SourceForge using the web interface, go to a designated page that shows the contents of the anonymous ftp repository and allows selecting files for a release.

SourceForge used to be an open-source project. With increasing popularity, the project was bought and became closed-source. From the last open-source branch of the platform have emerged a number of other software development platforms - namely: SourceForge Enterprise [19], GForge [20], BerliOS [21], Savannah [22], Savane [23] and LCG Savannah [24]. Apart from their license they are similar to SourceForge and are therefore not listed seperately in Table 2.1.

2.3.1 Features of existing software development platforms

Table 2.1 compares existing software development platforms. Besides the platforms that are compared here, we also looked at GoogleCode [25], SourceFubar [26], Codehaus [27], OpenSymphony [28], Java.net [29], Tigris [30],

	SF	Trac	Collabnet	Launchpad	Origo
API	×	(√)	(√)	√	√
IDE plug-in	×	×	×	×	√
Hosting closed-source	×	√	(√)	×	√
Open-source	×	√	×	×	√
Extendible architecture	×	√	×	×	√

Table 2.1: Comparison of existing software development platforms

Picolibre [31] and Seul [32]. We chose to compare the SourceForge-like platforms, Trac, Collabnet and Launchpad to Origo, because they are the most active and popular platforms. The others are either not maintained anymore or cannot be compared to Origo directly because they are not software development platforms as such.

Table 2.1 only lists the attributes where the different platforms are not offering similar functionality. Each platform shown offers some sort of configuration management (most of the time Subversion or CVS), they all encourage collaborative development by offering various ways of communication, like web pages, forums, issue management and project overviews. Each platform has team management features with rights and role management of varying granularity; all of them offer searching the managed information and for each of them a version that offers hosting projects for free is available.

We also looked at large open-source projects to find out what solutions they were using for hosting their development. In that we looked at Open Office [33], the Apache Foundation [34], the Linux Kernel [35], the Mozilla Foundation [36], KDE [37], Drupal [38], Gnome [39], Mono [40] and Sun Microsystem's [29] open-source projects; most of them use a custom solution that integrates a standard configuration management server, Open Office and Sun's other open-source projects use Collabnet and Mono uses MediaWiki [41].

Some general observations follow from the comparison in Table 2.1. Trac comes closest to Origo, except that it is not designed to host multiple projects on one instance. For that reason there is no support for exchanging information among different Trac projects. It also has an API, but only when enabled using an extension; hosting closed-source projects using Trac is possible, but a team has to setup and maintain its own Trac instance itself - there is no site offering Trac based project hosting off the shelf.

Collabnet has a limited API that is mainly offering ways to query and manipulate issues and it can be used to host closed-source projects as well, if one is willing to pay for a owned Collabnet installation - a site offering off

the shelf Collabnet hosting is Tigris [30].

Launchpad [42] has the best inter-project communication capabilities of all the compared platforms, but it is not extendible (closed-source) and has no API. The efforts taken by the Launchpad team are - like Origo - encouraging all tools used in development to offer an API.

None of the platforms offers plug-ins for existing IDE's and except for Trac the architectures of the platforms cannot be extended as they are not open-source or are not built in a way that allows extension.

2.4 Software development at ETH Zurich

In spring 2007, Benno Luthiger from the specialist department for open-source software at ETH Zurich carried out a survey on software development at ETH Zurich. The survey [43] had the goal to gain an overview on how software is developed at ETH Zurich. The survey was taken by 80 people.

The study shows that the infrastructure offered by ETH to support software development is insufficient. The interrogated developers say that the efficacy of their work would increase with better availability of supporting tools. The choice of tools that they are using in an ad-hoc way today are Subversion and CVS for configuration management (66%), web sites and Wikis for collaboration (59%). To track issues of the developed software Bugzilla or no tool at all is used in 80% of the cases. To distribute the binaries and to present information related to development web sites, Wikis and program documentation account for 59% of the cases. When asked on software development platforms that they are using 90% of them declare using no platform or Sourceforge.

2.5 Positioning Origo

In Section 2.3.1 the overview of todays software development platform shows in table 2.1 how Origo improves the state of the art. Origo has three major contributions: the platform offers an API; it hosts both open- and closed-source software development projects; and its architecture is ready for extending the platform with new applications. The offered features and the degree of integration corresponds to the needs of developers as analyzed above. Building an application composition middle ware using a peer-to-peer infrastructure is novel and proves to be practical, scalable and fault tolerant and solves application composition problems just like other existing frameworks.

Chapter 3

Architecture of Origo

Origo provides services like configuration management, issue tracking, release hosting and also an API that can be used to integrate Origo into other applications. The architecture that combines these services is the Origo Core. The Origo Core is the middleware and control layer of the Origo platform and constitutes the framework that can be used to also model other distributed service integrating platforms. Origo Core allows implementing use cases and controls the various Origo components accordingly. An example for a use case would be to following: The platform we build with OrigoCore allows publishing documentation online. This chapter presents the requirements for OrigoCore, shows how the framework addresses them and then details the usage of OrigoCore's API.

3.1 Requirements on the architecture

The goals of the architecture for Origo are to be open, modular and extensible. We want to build platforms that integrate several services. Adding a new service should not be difficult and extending an existing set of services with a new one should be possible dynamically.

Services can run in different environments, possibly on different machines. The framework should allow distributing services easily.

Whenever a service experiences heavier use than others it should be possible to balance the load on that service by means of adding several instances of that same service to the platform. The framework has to allow running multiple instances of a same service.

3.2 Framework

Origo Core is service oriented [44]. The services are provided by nodes and the core is the message bus and controller. Based on programmed use cases, the Origo Core controls the other Origo nodes by sending control instructions. The use cases are programmed as Eiffel classes. Control instructions are sent as asynchronous messages over a reliable message layer.

3.2.1 Nodes

The Origo Core framework provides functionality common to all nodes. The communication infrastructure implements reliable message passing and node discovery. When building a distributed platform using the Origo Core framework, only the node specific parts have to be implemented additionally. To facilitate this, Origoprovides several hooks, extension points and documentation.

Node discovery: As soon as an Origo node starts running, it sends a discovery message to the P2P network to look for core nodes. After receiving response, the node registers at all running core nodes. Each node stores information to contact each core node and each core node also keeps contact information on all known running nodes. Using these details, a node can send a message to a random core and each core can address messages to specific nodes. This approach allows scaling and dynamic extension or update, as there is no single point of failure and nodes can be added and removed at any time. Because nodes can leave the network, the discovery process is periodically restarted, with a default interval of 10 minutes.

3.2.2 Communication

The most important aspect of Origo Core is communication between the different services and applications forming a bigger platform. The services interact using each others data and exchange status information that is then reflected in the state of the entire platform. The exchange of data is multilateral, as one message possibly can be used for further processing by more than one other service.

Goals for Origo Core are to integrate services and applications that are running on different platforms and to scale well when need arises. These two reasons (platform plurality and scalability) combined with the nature of data exchange made us look into existing peer-to-peer framework implementations.

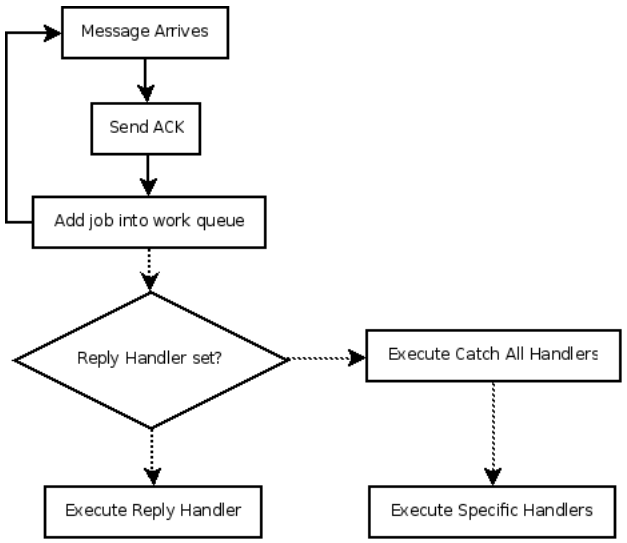


Figure 3.1: Receiving a message

With the JXTA [2] P2P specification standard we found a suitable framework. Applications like Collanos [45] show the possibilities of JXTA. the standard is implemented in Java and C and thus provides a way to interface to applications written in those languages. As Origo Core itself is implemented using Eiffel [46], we made an implementation [5] of the JXTA standard for Eiffel.

Using JXTA, Origo can propagate data through the platform to all applications and services that require a certain data item. The routing and all other protocols related to that data exchange are handled by the P2P infrastructure and take place in a platform and technology independent fashion, thus not excluding future extensions of the integrated platform with new applications, services or technologies.

The Origo Core framework uses a message-passing architecture [47]. Nodes exchange data as messages. The protocol for message transport guarantees that no message is lost, or received twice. This allows to send asynchronous messages reliably from node to node.

Threading: Messages are received in one thread. This receiving thread creates a job per message and adds it to a queue. Several worker threads process the job queue and execute all actions a use case foresees for a given message. This separation ensures that execution of actions does not block the communication layer (see Figure 3.1).

```
create l_msg.make
l_msg.set_reply_handler (agent (a_msg: O_MESSAGE)
do
  mutex.lock
  condition_variable.signal
  mutex.unlock
end)
mutex.lock
send_message_core (l_msg)
condition_variable.wait (mutex)
mutex.unlock
```

Listing 3.1: Synchronous messages

Main Event Loop: Sometimes, it is preferable to execute all actions in the main thread. This is necessary whenever if a non-thread-safe library is accessed in the use case (for example a database library). To allow this, actions can be registered to be executed in the main thread. As soon as a message for such an action is received, the call to the handler will be added to a job queue that is processed in the main thread.

The reliable message passing layer of Origo is implemented as a JXTA [2] service using VamPeer[5]. The reliability layer of TCP [48] is the model we took to implement our own reliable communication. Each message has a unique generated ID. After reception of a message, an acknowledgment message containing the ID is sent back. A receiver additionally keeps a list of arrived messages to avoid duplicate messages if there should be a problem with the acknowledgment message. This gives a reliability in the following sense:

- The sender knows if a message was received.
- Each message is received only once.

Unlike TCP, there is no guarantee about the order in which the messages arrive; as messages are self contained, asynchronous units, this is not needed.

Synchronous Messages: It is possible to simulate synchronous message passing with asynchronous messages. To do this, the sending thread is locked after sending the message and a reply handler is registered which will unlock the thread (see Listing 3.1). In most cases it is better to use `wait_with_timeout` instead of wait and handle the timeout case.

3.2.3 Using the Origo Core framework

Writing use cases: Use cases are created in the core node and are implemented as Eiffel classes. This allows very good flexibility and performance at the cost of not being able to change them without recompilation. Future work may be to extend the Origo Core framework with a scripting language that allows describing use cases.

Writing Client nodes: To create a new Origo node, include the Origo node library in your project. Your custom node should Inherit from the class `O_NODE_CLIENT` and it should implement the following features:

- `peer_name` – the name of the peer, must be unique in the Origo system
- `peer_description` – the node description
- `register_message_handlers` – handlers for general messages
- `register_*_message_handler` – handlers for specific messages

The framework provides an API call to retrieve information about the existing names from a live and running instance. Therefore it is possible to find a unique name for a new node that will be added to the instance.

Received messages are handled by message handlers which can be registered for messages of a certain type or for all messages (catch all). In addition it is also possible to register a reply message handler when a message is sent. Such a reply handler will then be called instead of the normal message handlers.

To send a message, the following features are available:

- `send_message_core` – send a message to a random core, this is used to start a use case, e.g. from an external API node
- `send_message_reply` – send a message back to its originating node

3.2.4 Design principles

Origo Core and the development platform using Origo Core follow a few design principles: Scalability, extensibility and integration.

Scalability: As mentioned before, Origo Core is built in a way that allows for good scalability and redundancy. It is possible to have multiple core nodes. For each use case a random core is selected by the peer starting the use case. After this, messages belonging to the same use case are sent as replies and therefore the same core node is used. This allows keeping track of the state in a use cases.

It is also possible to scale other nodes as all communication goes over core nodes which can distribute requests. For example it would be possible to have multiple storage nodes (see Section 3.4.3), each responsible for a range of data. As requests arrive at the cores, they would redirect the requests to the correct storage node. In a similar way, keeping a backup of a storage node is made easy as all requests that change data would be sent to both nodes and all data inquiry requests would be sent to either one of the nodes.

Extensibility: Extension of the platform means that a new service or application can be added to an Origo platform, without a complete rewrite of the platform or a complete redefinition of the involved databases and associated use cases. This is possible by adding a new custom node as described above.

We note, however, that some services or applications are more difficult to wrap as nodes to be used as extensions than others.

Integration: The third design goal – Integration – is achieved by providing a API construction framework within Origo Core. A platform built using Origo Core should expose API for all services and applications it consists of. Exposing an API with which a user interacts with the integrated services within a platform, simplifies management of changing service component interfaces. Another advantage is the possibility to build up a regression testing suite that uses such an API.

By offering an API Origo Core allows other applications to use a platform. For example – the Origo development platform does not interfere with the way you develop software. But once the system under development is ready to be released this can be done by using the API exposed by the Origo platform directly from within the tools you are using to develop your software system.

Integration means that by providing an API that is general enough, interactions coming from outside the platform are not only possible but encouraged. We are providing and XML-RPC [6] API layer. This makes it possible to integrate Origo API calls quickly into any scripting environment and development tool that you are using.

3.2.5 Origo instances

An Origo Instance is an integrated platform built using Origo Core. Our main goal was to build a framework, that allows combining external applications to become an integrated platform for developers.

In this Section we show how such an integrated platform can be built. We discuss the external applications that are selected, show how to integrate them into an Origo Instance. We then also show how an integrated platform can evolve over time and how a new external application can become a part of the platform.

The Origo development platform that we use to show how to build Origo instances is an online platform that developer teams as well as software users can interact with through various ways (see below).

The development platform is used to host the generated binary releases of software projects, it hosts the web site and the documentation of a project and it redirects to external sources of information (like web interfaces to configuration management servers for example). The platform integrates a bug tracking system, a configuration management repository, forums and blogs. Origo Core orchestrates the set of external applications and services. Figure 1.2 in the first Chapter shows the entire development platform. The P2P back-end is built with the Origo Core framework. The nodes inside the back-end provide access to the external applications and services (see Figure 1.3). The API layer is one of the ways to interact with the platform and will be detailed in the next paragraphs.

The services and applications used in todays software development process are subject to rapid change. A platform that wants to remain competitive and up to date must provide means for integrating emerging applications. Using the Origo Core framework together with the steps discussed above such an extension, evolution or adaption is accommodated.

Interacting with an Origo instance: Software development involves many different tools, services and applications. What remains common to the development process is the path from source code, that is being written to the publication of a software release.

Along this path all the different applications that are used for development come into action. The development platform that we are building with Origo takes a precise look at the different actions along the path and proposes automation and integration where possible.

The platform provides a number of interfaces for interaction. On top of the back-end itself resides a web interface that can be accessed with a browser. This web interface provides a user interface for all parts of the development

platform and for some of the functionality of the external applications – more on this in Chapter 6.

The second possibility to interact with the development platform is the exposed API. As discussed before, many actions along the path from the code to the publishing of a projects' release can be automated. Automation is only possible if the platform that is hosting the project provides an API that allows other applications from outside the platform to use it. Defining which API calls are exposed to the outside world is part of the task of the designers of an integrated platform solution that uses Origo Core.

The third way of interacting with the development platform is to use applications that interact with the development platform using the API. There is a number of development environments and other tools that use Origo API. The most popular ones are the plug-ins for Eclipse, VisualStudio and EiffelStudio. Others are being developed at the time of writing such as a widget for OS X or a makefile binding. For a complete list of applications using the Origo API refer to the web site.

The API that the integrated platform offers is the part of the system that allows integration of the platform into a development teams' process. By providing an API layer that is independent of the tools developers are using to program debug and compile their projects, the platform can be used by everybody. With the choice for XML-RPC [6] for the API a wide range of programming languages already offer frameworks to communicate with the platform. For those that do not provide ready made XML-RPC frameworks, the protocol is using HTTP as transport layer and support can thus be implemented quickly.

3.2.6 Dependencies

To avoid having to reinvent the wheel, a lot of functionality from external libraries is used for Origo Core. On the upside this makes it possible to share code with other projects and reuse existing code; on the downside we encountered a few bugs in the libraries that were quite hard to track down but could be fixed and therefore help to improve the used libraries.

- **EiffelStore:** an object-relational database abstraction library is used to access the database through ODBC.
- **EiffelThread:** is the multithreading library for Eiffel.
- **Eposix:** is a posix [49] wrapper library for Eiffel. Origo uses it to start the nodes as daemons.

- **Framework base:** is an utility library from the official Eiffel Subversion repository. We use its flexible command line argument parser.
- **Goanna:** is a web application library for Eiffel. We use its XML-RPC implementation and fastCGI binding in the API nodes.
- **Gobo:** Gobo provide free and portable Eiffel tools and libraries. We use various parts of the Gobo data structures and collection framework.
- **Log4e:** is a logging framework for Eiffel. We use it for logging in VamPeer and Origo Core.
- **Thread Extension:** a small library provifing helpful extensions to work in a multithreaded Eiffel environment. We use it in VamPeerto manage thread pools.
- **VamPeer:** the Eiffel implementation of the JXTA [2] P2P protocol which is used for the communication layer of Origo Core.

3.3 Front-end

The integrated platform built using Origo Core combines several external applications. Each of these applications can possibly have its own web interface, like for example a web front-end for a subversion repository. But also external applications that have no web interface themselves offer their functionality to Origo Core and the platform may want to provide its own user interface for managing these applications.

When combining a number of applications both with and without user interface to become one integrated platform it is neither possible nor desirable to leverage the entire functionality in the web interface. Successful web applications today present slim, but highly functional and fast user interfaces. The development platform aims in the same direction. The web interface presented to the user only allows a subset of actions to be carried out. To be able to use all functionality one has to use the platforms API. The subset of actions that can be made available in the web interface is not fixed. If use of the platform shows that a certain desired action is missing, it is possible to integrate it into the web interface as well. The design restriction that the web interface should remain slim and fast remains however.

The applications that provide a web interface themselves continue to do so in the development platform. The single sign-on functionality of Origo Core serves as gateway to them. In this case the platform only forwards

the interactions taken and the added value is that the authentication and authorization are taken care of.

We believe that developing applications not only involves a team of engineers, but should also involve users of an application. Users are the main source of feedback for a development team. The closer the user community and the developer community can be, the more feedback the developers get from their software users. This results in better software, that is maintained better and will again attract more users. In Chapter 6 the third version of the web interface of the development platform is presented.

3.4 Back-end

The back-end of the development platform is built using Origo Core. As mentioned before, Figures 1.2 and 1.3 show the different parts of the back-end. In this Section the node types involved are discussed in more detail. All node types relate to the development platform that is built using them, but their abstracted functionality is suitable for integration into other platforms built using Origo Core as well.

3.4.1 API node

The API nodes provide an XML-RPC [6] interface for various Origo Core services. It uses the XML-RPC interface from Goanna [50]. Each call sends a message to the Core where the corresponding use case is started. There are two operation modes for the API node. The internal mode provides some special services (like user signup, password handling, ...) that should only be available for calls executed by the front-end. Other wise denial of service attacks on the API of an Origo instance would be possible. The external API provides all remaining services and can be accessed once the API client is authenticated and authorized.

Goanna provides several possibilities to connect to the XML-RPC interface. The most efficient solution is to use a web server like Apache [34] or lighttpd [51] and use the fastCGI [52] binding. Figure 3.2 shows the connection between the clients and the Origo nodes with web servers in the middle.

For our development platform, the Lighttpd web server is used, as it provides better performance, better scalability and allows easy load balancing across multiple API nodes comparing to the other available web servers.

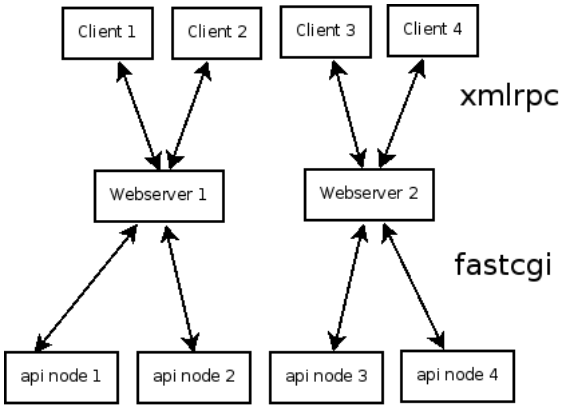


Figure 3.2: API web server connection

3.4.2 Build node

The Build node makes it possible to start a build of a hosted project and return the compilation result. At the moment it is implemented as a process call to an external script and the return value of the external script is returned as compilation result.

3.4.3 Storage node

All the data of Origo is stored in storage nodes. These nodes accept all kinds of access and modification messages. Internally, the storage nodes use a database to store and retrieve data. The storage nodes are split into an interface and a persistence layer. The interface layer handles incoming messages and and calls features on the persistence layer which performs the database access. This keeps the interface clean of database specific code.The storage nodes store specific information about the state of the development platform and also about the configuration of the involved services and applications. The storage nodes constitute the model of the MVC pattern used for Origo Core. They can be accessed from the entire P2P network within Origo Core.

For database access EiffelStore [53] and a small wrapper, to simplify access to query-results and error handling, is used. The used database is MySQL [54] and we access it using the ODBC [55] binding of EiffelStore.

3.4.4 Configuration node

To use external tools like Subversion [56] or an email server, a way to generate configuration files and execute configuration scripts is needed. Configuration nodes are the back-end part that is used for such purposes.

To create work items for Subversion commits, a hook script is used which is automatically called by Subversion after a commit. This script then retrieves additional information like the changes of the commit and makes an API call on the internal API to register an event of project life and consequently trigger notifications of all concerned developers and users. The script is a based on the `mailer.py` from the Subversion delivery.

The mail node enables the development platform to send an email. Email notification of events in project life as well as maintenance alerts or user interaction happens on the development platform through email. The mail node is implemented by using the SMTP [57] support of EiffelNet [58] to call a local or remote mail server which handles mail delivery.

3.4.5 Use cases

Log in: The log in of a user is an example for a simple use case (see Figure 3.3) where only few nodes take part. A client executes an XML-RPC call on an API node. This node sends a message to the cores which forward it to the storage nodes that lookup user credentials and – upon success – return a session. The session is forwarded to the API which returns it as the result of the XML-RPC call.

Password reset: A more complex use case is the resetting of a password. Additionally to the API, core and storage nodes also a configuration and mail node are involved (see Figure 3.4). First a new password and the email address is retrieved from the storage nodes, then an email with the new password is sent to the user and all configuration files for external applications and services are generated anew.

3.4.6 Authentication and autorization

In the process of building an integrated platform with Origo Core, a number of external applications and services are combined. All of them provide some form of user authentication, user management and role distribution. Users of such a platform should not need to remember all usernames and passwords to be able to interact with the platform. To address this, Origo Core has a authentication and authorization functions built into the framework.

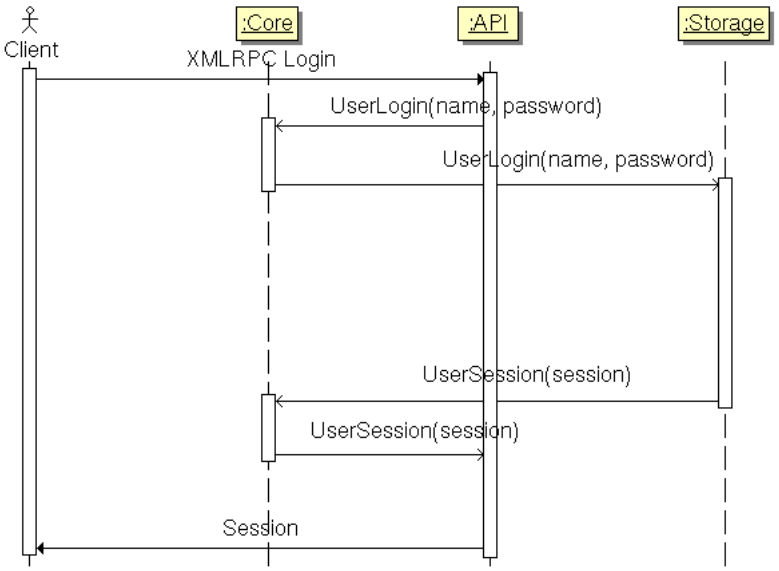


Figure 3.3: Log in sequence diagram

Using core and storage nodes, the framework stores all usernames and credentials required by the external applications. Using the web interface passwords and usernames can be registered with the platform. This allows to provide single sign-on functionality and thus interacting with the integrated platform is easier. The platform is aware of the external applications and their authentication mechanisms. If a user interacting with the platform has not provided a username and password for one of the applications or services, the platform handles automatic creation of user credentials.

To have finer control on allowed actions of Origo users the framework provides authorization schemes based on roles. Origo Users can have different sets of rights according to their role; – the development platform uses few different roles in order to keep things simple, but this can be extended without changing the framework architecture. The external API allows users to log in to the development platform. During log in credentials are checked by storage nodes and a session is generated which represents this user. API calls require this session. A session maps to an Origo user but the same Origo user can have more than one session at a time. After a certain time of inactivity a session expires and becomes invalid.

To avoid brute-force attacks but still allow external applications to use the API, instead of a login with username and password, a login with a user and an application key is provided. Each external application and each Origo user can generate a custom user key which can then be subsequently used to

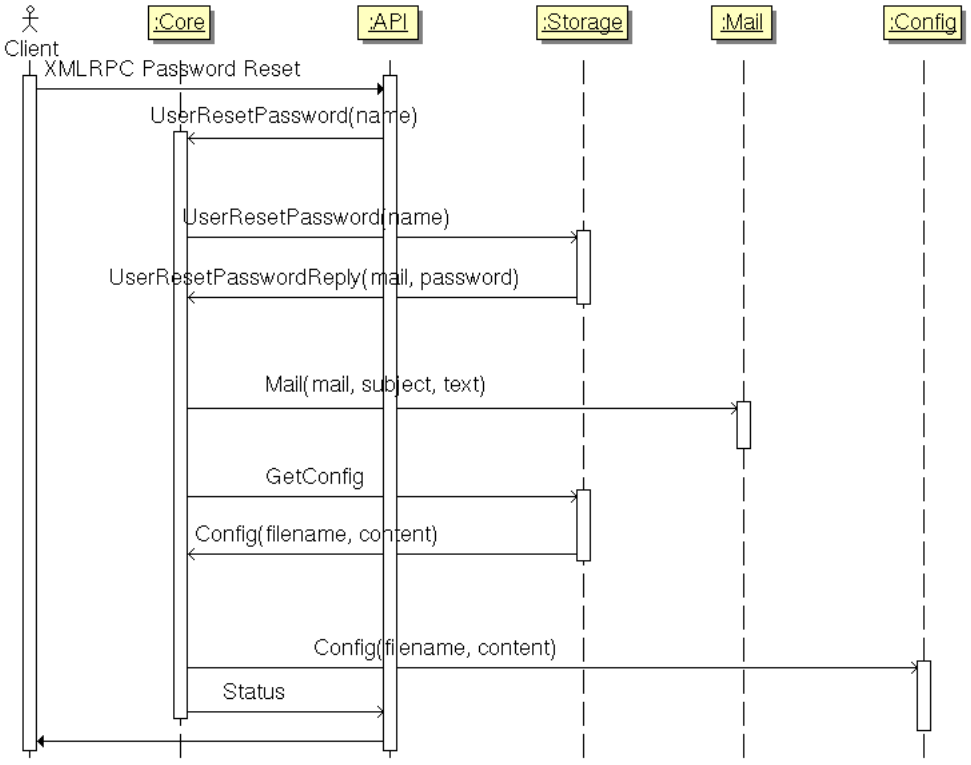


Figure 3.4: Password reset sequence diagram

log in and obtain a session. The session received by a login with username and password or with an application key are equal.

Authorization in Origo is based on several access groups. Each group can be either global or project-specific. Examples of global access groups are:

- Origo User
- Origo Administrator

Examples of project-specific access groups are:

- Project Owner
- Project Developer

The group structure is flat: groups can only contain users and not other groups.

For each action that requires authorization, a policy defines which access group is needed. For example to add a new member to a project as project developer, the action has be executed by a project owner of this project or by an Origo administrator.

3.4.7 Deployment init scripts

On Unix and Linux, init scripts are in charge of starting background daemons. Init scripts provide the possibility to start, stop and restart such a daemon. The init scripts also read some configuration which allows to configure things like log level or listening interface.

As the Origo nodes already provide an option to daemonize, the scripts are relatively straight forward and could be created based on a skeleton provided by the Debian distribution. Before starting, a check is made if the process is already running, otherwise the process is started. Similar during the stop, a process with this name is searched and first a TERM [59] signal is sent, if this does not stop the process a KILL [59] signal is sent to ensure that the process quits. This is mostly done by using the start-stop-daemon [60] tool. The rendezvous node is written in Java it does not provide a way to daemonize itself, therefore the start-stop-daemon [60] tool is used to detach the process from the console and put it into the background. This also writes a file which has the process ID in it and which is used to stop the process, similar to the way an Origo node is stopped, first by sending a TERM [59] signal and then a KILL [59] signal.

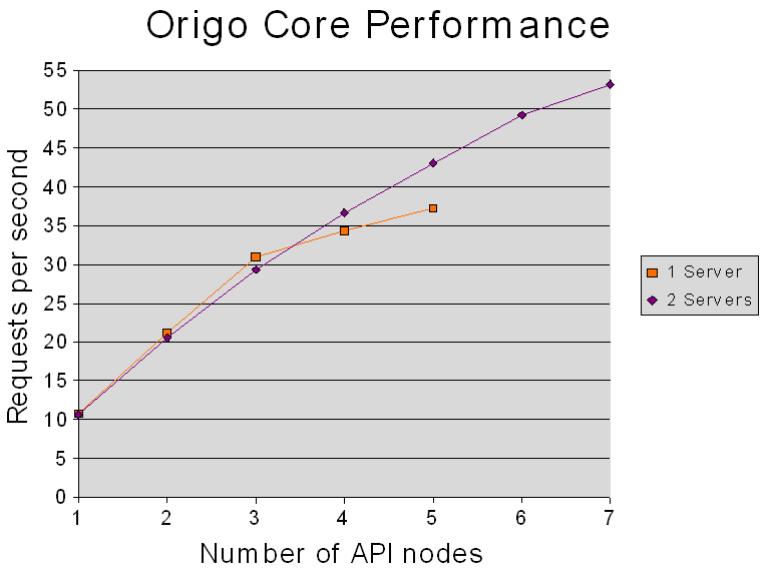


Figure 3.5: Benchmark results

3.5 Performance

We did various performance benchmarks and profiling to assess scalability and find out where the performance can be further improved.

Origo Benchmarking: to test overall performance and scalability of the platform, a tool to issue large numbers of API calls which then trigger Origo messages and actions in nodes is used. This is a realistic benchmarking scenario as almost all use cases of Origo Core start on the API. To do the testing, the ab tool from Apache [34] is used.

Figure 3.5 shows how Origo scales with one server and with two servers using multiple API nodes. Some more performance results can be seen in the Origo project of the platform [1].

It is interesting to note that if Apache is used, the performance is worse and does not scale at all. If Lighttpd is used the performance is better and also scales better. We can see that having multiple API nodes on the same machine scales well and also having multiple machines does scale very well.

[1] http://origo.ethz.ch/wiki/performance

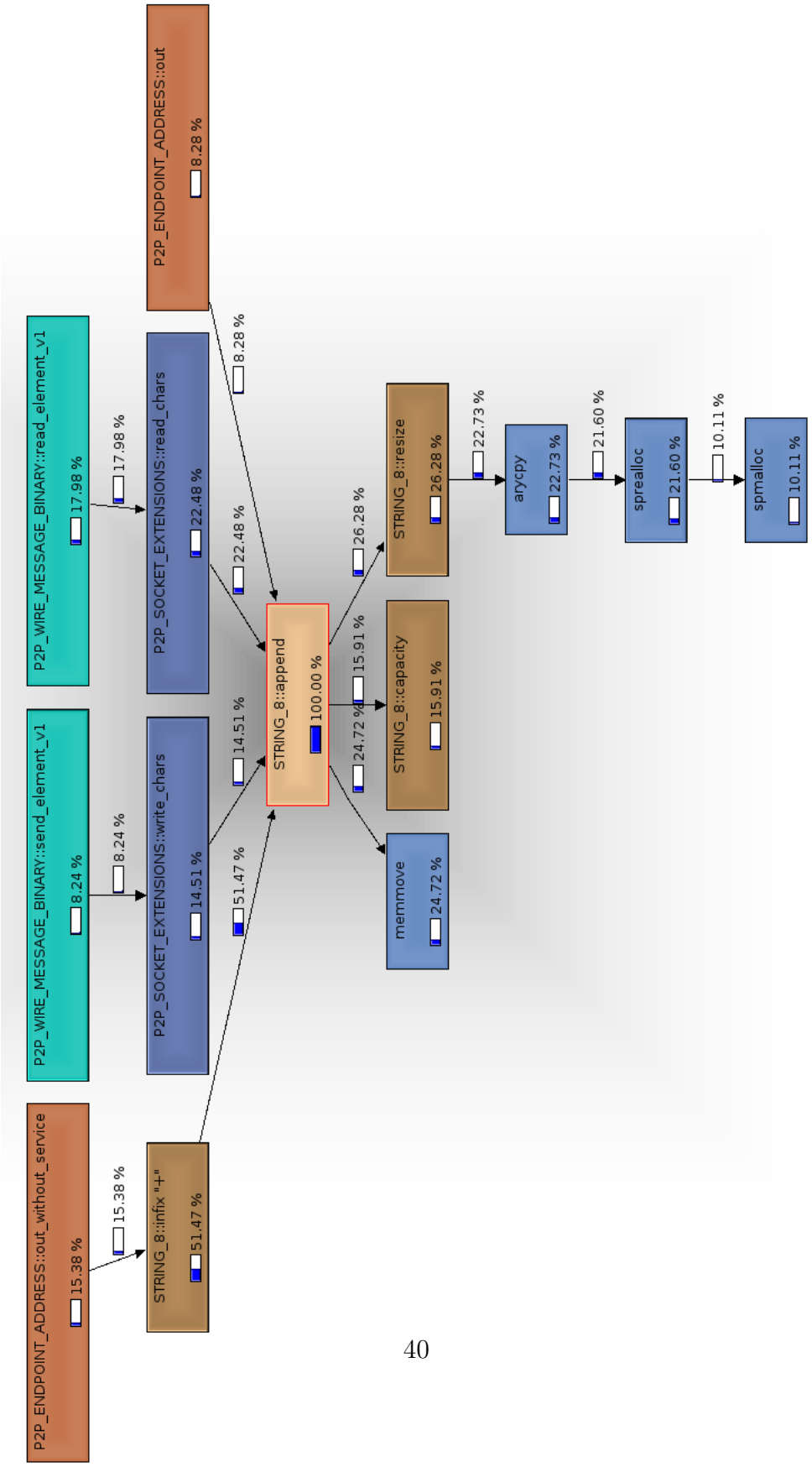
3.5.1 Profiling with Valgrind

To find out where most computation time was spent, we use callgrind from the Valgrind [61] suite for profiling. As valgrind works on the C level, we built a small tool retranslating the C names back to Eiffel names [62]. This data is analyzed with KCachGrind [63] which generates helpful graphical representations. A sample of such a graph can be seen in Figure 3.6. The graph shows the call trace and the relative time spent in the features relative to the selected feature.

3.5.2 Performance estimation

As the performance tests show, the limit for the current setup is around 50 log in requests per second. If we consider "login" as a representative request this can be used to estimate how many users the current setup of Origo can serve. This does not include the load of external programs like the Subversion repository backup processing or the web site itself.

We estimate that on average a user that visits the web site executes an API request every 20 seconds. The other time is spent reading pages and doing web site requests that do not execute API calls. So if we can handle 50 requests per second and an average online user executes a request every 20 seconds, that results in the possibility to have about 1000 concurrent users online. Experience from other projects show, that in online projects, no more than 10% of all registered users are concurrently online. Therefore the current setup should be able to serve about 10000 registered Origo users.


P2P_ENDPOINT_ADDRESS::out_without_service
15.38 %
15.38 %
P2P_WIRE_MESSAGE_BINARY::send_element_v1
8.24 %
8.24 %
P2P_WIRE_MESSAGE_BINARY::read_element_v1
17.98 %
17.98 %
STRING_8::infix "+"
51.47 %
51.47 %
P2P_SOCKET_EXTENSIONS::write_chars
14.51 %
14.51 %
P2P_SOCKET_EXTENSIONS::read_chars
22.48 %
22.48 %
P2P_ENDPOINT_ADDRESS::out
8.28 %
8.28 %
STRING_8::append
100.00 %
24.72 %
15.91 %
26.28 %
memmove
24.72 %
STRING_8::capacity
15.91 %
STRING_8::resize
26.28 %
22.73 %
anycpy
22.73 %
21.60 %
spreallocl
21.60 %
10.11 %
spmalloc
10.11 %

Chapter 4

Communication Infrastructure

In this chapter we describe what kind of communication is needed for Origo.

4.1 P2P systems

Peer-to-peer systems have become very popular in the last few years. They allow users to share resources (such as calculation power or information) in a distributed and decentralized way. Peer-to-peer (henceforth called P2P) technology is different from the client-server model which relies on one central server fulfilling all tasks for the clients. In a P2P system every participant is considered equivalent. An overview of P2P systems is given in [64]. The survey is based on the most important systems and summarizes the key concepts. In P2P applications, we often want to perform common tasks like discovering other peers, sending and propagating messages and sharing information. While many P2P applications implement their own solutions, there also exist frameworks which provide an API for most of these common tasks.

One of the design goals for Origo Core was to be scalable and dynamically extendable. We wanted to try to achieve this goal by using a P2P based approach for building the back-end of Origo with Origo Core. For Eiffel, there was no P2P framework available. Therefore we implemented one based on an existing specification: JXTA. We call our framework VamPeer.

The main purpose of our implementation is to have a P2P framework for Eiffel, so we can build Origo Core using it; VamPeer is the communication infrastructure for Origo nodes.

Chapter 3 introduces Origo Core and the development platform built using it. The P2P library is standalone and does not depend on Origo Core – it can thus be used to build other P2P based applications in Eiffel. VamPeer is generic and can be used for many other purposes than Origo Core.

JXTA is an open-source P2P framework specification created by Sun Microsystems [65]. It is one of the most mature platforms in its field. JXTA (pronounced *juxta*) is composed of several modules each implementing a specific JXTA protocol (for example the discovery protocol). The protocols are XML based. The decision to use JXTA was quite simple since it is the only platform independent P2P framework we found. Porting JXTA to Eiffel gives us the advantage to remain compatible with other JXTA implementations. While the reference implementation is written in Java JSE, there also exist other bindings written in C, JXME and other languages. Most of the other smaller bindings are not yet ready for production use.

4.2 Criteria for choosing a P2P framework

One of the most similar other frameworks is *Jini*. Jini is not suited for our purpose since it only runs on Java. Furthermore, it uses a central server to locate network services in contrary to JXTA which follows a completely decentralized P2P model. There is also another framework called *OogP2P* but since it is a simple research project and is not maintained anymore, we did not consider it for implementing Origo Core. Besides frameworks, there are plenty of P2P networks defining protocols for sharing content (for example *GnuNet*). Research projects such as *Chord* usually provide special algorithms for a distributed hash tables. These projects do not meet our requirements because they focus more on sharing information and lookup algorithms instead of application construction facilities.

We choose to implement an Eiffel port of JXTA because JXTA is used more and more in modern applications (for example *Collanos Workplace*). A short overview of the mentioned P2P frameworks and protocols is available at the VamPeer web site[1].

4.3 JXTA Concepts

Before we describe the design of VamPeer, we give an introduction to JXTA. We show how the JXTA protocols are designed and present an overview of the specification. The full specification [66] covers the basic ideas around JXTA and specifies the messages which go over the wire. For more details on semantics, we recommend the literature or looking at the reference implementation source code.

[1] http://vampeer.origo.ethz.ch

The paper [67] focuses on release 2 of the JXTA protocols and gives a good overview. The free book by Brendon Wilson [68] explains JXTA with many Java examples and [69][2] goes more into the details of the Java reference implementation. Unfortunately, the books are slightly out of date.

We will first look at peer groups, define JXTA IDs, advertisements and then introduce the different services and protocols available. Afterwards, we present the overall P2P networking infrastructure. We do not remain only at the JXTA specification level but also show a few design ideas used in JXTA JSE, the Java reference implementation[3].

4.3.1 Peer groups

A peer group is a compound of peers agreeing to run the same set of services[4]. When a peer joins a peer group, all services needed should be loaded according to the peer group's specification. Thus, a peer is always a member of at least one group as there would not be any running services at all otherwise. A peer may belong to more than one peer group. The super peer group which is loaded first is *usually* the world peer group (WPG). All other groups are direct or indirect children of the WPG. This is because only one peer group can perform the actual handling of network traffic. The specification does not explicitly mention a parent-child relationship among peer groups, but it is handled that way in JXTA JSE and also in VamPeer.

4.3.2 World Peer Group

The WPG is defined as the peer group in which all JXTA peers reside, even if they are not communicating with each other. The WPG is somewhat a special peer group which is automatically loaded and may not support all services. In the C implementation, there is actually no explicit WPG whereas VamPeer uses one for configuration purposes only. The WPG's ID reads `urn:jxta:jxta-WorldGroup`.

4.3.3 Net Peer Group

The WPG has only one direct child: the net peer group (NPG) which is now a true normally running peer group configured with all JXTA services. When

[2] www.samspublishing.com provides a free sample chapter: "Java Implementation of JXTA Protocols"

[3] JXTA JSE is currently available in version 2.4.1, see [65].

[4] A service is a set of features following a specification either made by the JXTA project or the user, see Section 4.4.

talking about the NPG, we usually mean the *public* net peer group with the ID `urn:jxta:jxta-NetGroup`. It should only be used for development and testing purposes (as long as no other group has been created). Sun provides a public infrastructure for this peer group – unfortunately, the servers have been unavailable or under heavy load for the last couple months[5].

When using JXTA as a framework for a custom P2P application, one should create a new *private* NPG so that the application will not get in contact with peers from other applications[6]. It has its own network.

Other peer groups are children of the NPG. Although services can be shared among peer groups, one should have good reasons to split the application into several groups as peer communication is only possible within the same group.

4.3.4 IDs

We already mentioned IDs for peer groups. Also other entities in JXTA have an ID: peers, modules, advertisements and other resources. A JXTA ID must be a complete identifier referring to a unique resource. JXTA IDs follow the URN format (see [70]) with the namespace `jxta`. Additionally, the URN namespace specific string is prefixed with a format ID announcing how the ID is formatted. The general form looks like this: `urn:jxta:format-specificid`

Although the format is written explicitly, one should never make any assumptions about the ID format. Of course, it is allowed to optionally gather some information from an ID when the format is recognized.

4.3.5 UUID format

Most IDs are in the JXTA `uuid` format. They are in hexadecimal form representing 1 up to maximum 64 bytes. The last byte (the last two hex digits) always specifies the type. Each type stands for its own schema. Usually, the number contains one or two UUIDs[7], each 16 bytes long. Table 4.1 shows the defined types for the `uuid` format and all the information they contain.

[5] This downtime being cumbersome, it is no problem as one can easily setup an own rendezvous/relay server, see Section 4.4.

[6] This is not a guarantee however; every JXTA peer may connect as long as it knows the address and has a purpose.

[7] A "Universally Unique Identifier" (UUID) is a random number (containing also time information) meant to be universally unique (the probability to create two same UUIDs in the same context is very small). They are fully defined in [71]. You may prefer the corresponding RFC [72] which describes also the UUID format but focuses more on UUID as a URN namespace.

Sample IDs look like this:

– Peer group ID:
`urn:jxta:uuid-822A7C9E6B804759870B81B10070E9C9\`
`596162616461626 14A7874615032503302`

– Module class ID:
`urn:jxta:uuid-261F502615134AA99FDC99E3751E6B8505`

ID byte	*Type name*	*Information contained*
01	Codat ID	Group UUID, Codat UUID, Codat Hash
02	Peer group ID	Group UUID, Parent group UUID
03	Peer ID	Group UUID, Peer UUID
04	Pipe ID	Group UUID, Pipe UUID
05	Module class ID	Module UUID
06	Module specification ID	Module class UUID, Specification UUID

Table 4.1: UUID ID types in JXTA IDs

4.3.6 Advertisements

Another important concept in JXTA are *advertisements*. An advertisement is an XML document describing any kind of resource. It contains metadata. The JXTA protocols are then used to transport and share advertisements with other peers. For peers, peer groups and all the other JXTA entities, there are advertisement schemes defined. It is the JXTA way of creating new types of advertisements for each used entity. In a file sharing application for example, one would create a file advertisement containing a codat ID[8], the name of its owner, the creation date and the ID of the peer that hosts the file.

An advertisement has an expiration time, to ensure that no old advertisements are passed around. As advertisements cannot be withdrawn or deleted on remote peers, one should not set the expiration time too high; default expiration is two hours, but implementations may vary. When resources are valid for a longer time than the expiration time, the advertisement has to be recreated and published again. Below we show two advertisement types to better illustrate their purpose.

[8]A codat is just a container for any kind of data, for example file content.

4.3.7 Peer Advertisement

One of the important advertisements is the one describing a peer. This is for example used to discover peers. It contains the peer ID, the peer group ID and an optional peer name as well as a description. Each service may additionally add a service parameter with some specialized information. Listing 4.1 shows a sample peer advertisement. The service parameter contains the information for the endpoint service: a route advertisement that advertises the physical endpoint address (IP host and TCP port).

Listing 4.1: A sample peer advertisement for a peer in the public NPG

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:PA xmlns:jxta="http://jxta.org">
  <PID>urn:jxta:uuid-59616261646162614E50472050325033\
    1A227980E5924E80A3FD8ECD73D4C31803</PID>
  <GID>urn:jxta:jxta-NetGroup</GID>
  <Name>My sample peer node</Name>
  <Desc>Development test peer</Desc>
  <Svc>
    <MCID>
      urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE0000000805
    </MCID>
    <Parm><jxta:RA><Dst>
      <jxta:APA>
        <EA>tcp://129.132.105.170:32725</EA>
      </jxta:APA>
    </Dst></jxta:RA></Parm>
  </Svc>
</jxta:PA>
```

4.3.8 Peer Group Advertisement

The peer group advertisement announces the existence of a peer group. It contains the group ID, the module specification ID[9] and optionally a name, description and service parameters.

4.4 JXTA services

Until now, we have just introduced *peer groups* and the general terms *ID* and *advertisement*. In this section, we will look at *services* and what JXTA protocols they are providing. JXTA is modular. Each feature or protocol is available as a service module. Thus, a peer group can disable unneeded modules.

Module definition: JXTA modules can be loaded dynamically. This means that a JXTA peer can theoretically load module code from another peer. For specifying and identifying modules, several module advertisements exist: A class of modules providing the same local behavior and API is identified by a unique *module class ID* (MCID) and announced with a *module*

[9]See Section 4.4 for an introduction to modules and their specifications.

class advertisement. Specifications for a module class are identified with a *module specification ID* (MSID). Its *module specification advertisement* includes a version number and a URI where a human-readable description can be found. The specification focuses on the remote behavior and the protocol. All modules implementing a module specification can be advertised with a *module implementation advertisement*. This specifies the targeted environment and may provide the entire code or a package name and a description where the code can be fetched. The three module definition layers allow to have various specification versions for a single module class and also any number of module implementations for each environment.

We will now present the main services we have focused on in VamPeer:

Endpoint service: The endpoint service is the core service. All messages coming from the network (through the transport modules) are redirected to reach here and are then sent to other peers. The purpose of the service is to route the messages to the services that are interested.

Endpoint messages: The endpoint service is dealing with *endpoint messages*. Each transport module must be able to send them over or read them from the wire. A message is basically just an ordered list of key/value pairs (elements). The key is restricted to a namespace whereas the empty namespace and the `jxta` namespace are predefined for user respectively JXTA internal purposes. You may have any number of namespaces and keys. An element may additionally define a MIME type[10] and a signature element which is rarely used. The endpoint service adds some elements to outgoing messages, for loopback detection and for addressing.

Endpoint addresses: The source of the message and particularly the destination address must be defined: an *endpoint address* can be used for various forms of addresses. The string format looks like this:

`protocol://address/service/param`

The *protocol* part specifies the transport module to use: `TCP`, `jxta`, `http` for example. The *address* is protocol specific: For TCP and HTTP for example, it is of the form: `ip:port`, for JXTA it is a peer ID in URN format. For destination addresses, *service* defines the final service; *param* is an optional parameter for the given service.

[10] See [73] for a general introduction to MIME types and [74] for the XML type, which is used in JXTA messages.

To send messages to a peer, one may either create a destination endpoint address directly with the destination's IP and port number or alternatively just set the peer ID, which is the preferred way (because dealing with IP addresses is discouraged in higher level services). It is the *endpoint router*'s task to resolve the peer ID to the real endpoint address as described in the next section.

Transport modules: The *transport modules* are responsible for sending endpoint messages to another peer and for reading incoming messages from the network. Therefore, they register themselves as available transport module in the *endpoint service*, each one for its own *protocol*. They provide support for sending a single message to a peer, to ping a peer (looking if the remote peer is online) or to propagate a message. Each transport specifies its own wire representation. There is no required transport module and protocol but the low level transport TCP is usually enabled together with the HTTP transport. It is also be possible to send messages via SMTP or any other protocol. TCP transport is simple and fast, HTTP has the great advantage to pass through firewalls since most of them allow HTTP traffic.

The transport modules do not guarantee message delivery even when TCP is used. This is very important. The original message sender cannot be sure that his message has arrived at the destination. The message transport is usually not secured except for the `jxtatls` transport which uses TLS to encrypt data. It is based on top of the *endpoint router* to provide a secure path from the source to the destination peer.

Endpoint router: A special transport is the *endpoint router*. It is not used to transport messages over the wire but to route messages with a peer ID as destination address to the correct gateway. To do so, it rewrites the destination address and passes the messages again to the endpoint service, which then sends the message using a real transport. A message cannot be sent directly to the peer because there is no direct connection to it. *Router peers* can forward messages to other networks. The task for this module is to query for routes and to send the message to the first route gateway. It will try to connect using the fastest transport module available (whenever a connection to that peer is already opened, the related transport is considered to be the fastest).

When a peer, is configured to act as *router*, it accepts and forwards messages from other peers. It also maintains a route cache to be able to handle messages to be routed faster, without always having to first seek for routes.

Rendezvous service: The *rendezvous service* is used for propagating messages through the peer group and/or the local network. This fundamental service is used for sending queries to all peers.

Rendezvous lease protocol: A rendezvous client has to subscribe to a rendezvous server to be able to send messages for propagation and also to receive propagated messages. Therefore, a lease protocol is defined which manages this. When a client peer joins a peer group, it tries to contact to a rendezvous server. There are several ways to find one:

- The configuration of peer contains some endpoint addresses (addressing the peer directly with TCP or HTTP).
- The platform configuration of the peers specifies a seeding URL where a rendezvous server list is published. This is the most common and simple way for publishing rendezvous servers. It is also very useful for maintenance because only one list has to be updated to point all the new peers to other servers.
- As soon as a peer establishes contact with other peers, it may send them discovery queries for rendezvous peers. It may also cache rendezvous advertisements[11], so it does not have to find new servers everytime when starting up.
- It is also likely, that other peers reside in the same local network. Therefore, a peer may send a discovery query via multicast to the local net.

Once a rendezvous gets a lease request, it may send back a lease granted message – a lease with restricted validity for, usually 30 minutes. During this time, messages are propagated to the subscribed peer which is allowed to send a propagation message to the rendezvous. As leases are not valid for long, a client has to send lease renewal requests until it gets another lease. Renewal and initial lease requests look the same. JXTA JSE asks for renewal when the first half of the lease has elapsed. When a peer leaves a peer group, it should send a lease cancel message, so the rendezvous does not try to propagate messages to that peer anymore. A rendezvous client should only be registered with at most one rendezvous and should always send a lease cancel message to peers which send propagation messages without being the rendezvous in use. For further details on lease messages, please consult the specification [66].

[11] A *rendezvous advertisement* promotes rendezvous server capability of a peer.

Message propagation protocol: The *message propagation protocol* is used to propagate messages. It adds a message element[12] which has an XML document containing a unique message ID, a TTL, a path and the name of the final destination. The protocol ensures, that duplicated messages are discarded as well as messages that are too many hops away from the source peer. This can be done by looking at the message ID and the TTL respectively. Every hop decrements the TTL value, so the message can be filtered out when the value reaches zero. The protocol is also responsible for loop detection. Every hop adds its peer ID to the message's path and detects when a message already passed by earlier. Based on the given service name, the rendezvous service is able to pass the message to the correct service, for example by using the endpoint service. Unlike older JXTA versions, a rendezvous service should not repropagate every incoming message and thus flood the network. Each service decides individually whether to re-propagate a message or not – of course only as long as the peer is a rendezvous server. Hence, a rendezvous server should have some knowledge about the network and which peer may have which information. It is able to direct messages only to those peers that may have use for the propagated message. Of course, this heavily depends on the actual service and the type of the message.

Peerview protocol: As mentioned earlier, rendezvous servers need a good knowledge of the peer network infrastructure. They also need to stay in contact with other rendezvous servers to share propagated messages because there may be any number of rendezvous (not just one) and propagated messages are expected to reach eventually every node in the peer group (not just the subscribed peers of an owned rendezvous). To manage and share this knowledge, the *peer view protocol* is used.

Resolver service: The *resolver service*[13] is the first user of the rendezvous service as it has to propagate queries. Its task is to provide a query-response system. It is able to recognize received responses to a sent query by adding meta information to queries like a handler name and a query ID. It attaches the route information of the peer, so remote peers have the possibility to respond even if they do not know the querying peer. The actual query can be any string. The resolver service creates an endpoint message and combines the query together with all the meta information to a XML document as shown in Listing 4.2.

[12]The element name is the peer group ID prefixed with `RendezVousPropagate`.

[13]The service name is a little confusing: it means actually to resolve queries to responses.

Listing 4.2: A sample resolver query XML document

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:ResolverQuery xmlns:jxta="http://jxta.org">
  <SrcPeerID>urn:jxta:uuid-59616261646162614E50472050325033
     \
    1A227980E5924E80A3FD8ECD73D4C31803</SrcPeerID>
  <HandlerName>BeerFinder</HandlerName>
  <QueryID>1</QueryID>
  <HC>0</HC>
  <Query>Got a beer?</Query>
  <SrcPeerRoute><jxta:RA><Dst>
    <jxta:APA><EA>tcp://129.132.105.170:32725</EA></jxta:APA
       >
    </Dst></jxta:RA></SrcPeerRoute>
</jxta:ResolverQuery>
```

The hop count, which is incremented on each hop, ensures that the query is not sent to far away. Although the rendezvous service already performs a similar check, we have to do it again because the resolver does not always need to use the rendezvous propagation mechanism. Queries and responses can be propagated or sent directly to a specified peer. The resolver service not necessarily depends on the rendezvous service but would of course be limited to local network propagation and single message dispatching in a situation without rendezvous. A resolver response looks basically the same as a query. The actual response is a string. There is no hop count as it does not make any sense here. Also the source peer ID and route are dropped but there is a response peer ID, so the recipient knows from which peer the response originates. A client service sending a query should be able to register a listener for related responses, so it does not have to check itself if the response matches the query. That is one of the main tasks of the resolver service.

4.4.1 Discovery service

While he resolver does not maintain much data and only serves as an intermediate message layer for other services, the *discovery service* is a fundamental part in JXTA and much information passes it. The discovery service deals with all sorts of advertisements, so it knows about all peer resources (as advertisements are promoting resources). It serves as an advertisement storage and it is also responsible for finding remote advertisements and for letting other peers know about the locally stored ones.

Discovery queries and responses: Whenever a service needs an advertisement, it does a local discovery query which is equivalent to a storage lookup. Generally, we distinguish between peer, peer group and other advertisements, so we specify the advertisement type in a query. A query may restrict the search additionally with a key and a value name. The key is a XML tag name. It is allowed for the value to contain the wild card character – * – in the beginning and/or at the end. The number of answers may be limited by setting a threshold. Querying remote peers is actually the same but one may choose to send a query to a single peer or to propagate the message in the group. In both cases, the resolver service is used for sending the messages. The discovery query is an XML document specifying query type, key, value, threshold and optionally the source peer advertisement. Looking at the final endpoint message, we see a resolver element containing a resolver query XML document which contains the quoted discovery query XML data[14]. A peer is not obliged to respond to any remote discovery query. Peers that have sent a query should expect no, one, or multiple responses. They cannot expect that the threshold is respected, neither as minimum nor as maximum. A discovery response message can contain several matching advertisements. Additionally, it can also contain the responding peer advertisement. Discovery responses are not only used to respond to queries; but it is also allowed to publish advertisements to other peers, especially the rendezvous server, using a "response" message. To feed local storage with advertisements, one just publishes them locally.

Shared Resource Distributed Index: As we have already seen, a rendezvous server will not propagate every message. In the case of discovery queries, the rendezvous makes use of a *shared resource distributed index*, henceforth called SRDI (see [75]). The SRDI is an advertisement index containing certain keys and values together with peer IDs enabling the rendezvous to lead queries to peers which should have matching advertisements. This monumentally reduces network traffic as peers that do not have the needed information are not queried. Note that the SRDI does not contain the entire advertisements but has some important keys and their values for every advertisement a peer has. But how does the rendezvous maintain its SRDI? Every edge peer sends its SRDI to its rendezvous. When newly joining the group, it sends the full index. Later, it sends regularly (for example every minute) a SRDI delta, that means only the key/values for newly discovered, created or updated advertisements. When a rendezvous lease is canceled,

[14]The discovery query is quoted because the resolver service currently expects a simple query string.

the peer's SRDI entries are removed automatically by the rendezvous. There exists a generic SRDI XML document used for pushing SRDI entries to the rendezvous. It contains also a TTL, so an SRDI entry is not valid forever (like the advertisements themselves). The current JXTA implementations do not index each XML tag of every advertisement. When creating a new advertisement type, one should specify which elements are important and should therefore be indexed. When speaking in database terms, one should at least index the primary key attributes.

4.5 JXTA's P2P infrastructure and peer roles

To bring some clarification into the partly insufficiently introduced peer roles, we would now like to show a short overview of the entire JXTA P2P infrastructure: Although JXTA may use central rendezvous server lists when starting up, we can definitely see the JXTA structure as a *true* P2P system. It does not rely on central servers for any core task and uses the P2P structure for all purposes. However, there are various peer roles in the network:

Edge peer: Whenever traffic or CPU power is expensive, an *edge peer* is surely the right role for a peer. Such a peer heavily relies on other peers and consumes parts of their attention. Most peers would actually choose this kind of role.

Rendezvous peer: A *rendezvous peer* is providing a rendezvous server and enables edge peers to make contact with other peers. In a JXTA network, we need at least one rendezvous server because we usually want to discover other peers and do not have the physical locations hard coded of other peers that we want to communicate with.

Router peer: A *router peer* enables peers to communicate with others to which they cannot connect directly. This is used for peers behind a NAT gateway or a firewall. Therefore, router peers may have to manage all their clients message traffic in one or both directions. Note that peer roles may dynamically change. For example, a peer which cannot find any rendezvous server could automatically become a rendezvous. This is of course adjustable in the platform configuration.

4.6 VamPeer Design

Keeping our goal in mind, we focus on the essential parts needed for Origo Core porting *all* JXTA protocols to Eiffel too big a task for our setting. See Section 8.2 for future work on VamPeer and how the library can be extended to add missing features.

As we are implementing JXTA for Eiffel with regard to support the network layer of Origo Core, we state the following requirements:

1. Origo Core peers may communicate among each other without being disturbed by messages from other peer applications.

2. An Origo Core peer is able to advertise its existence to the peer group and can also discover other peers, for example a core node.

3. An Origo Core peer is able to send messages to other peers. A message may contain data of any type and length.

Mapping these ideas to the JXTA world, we specify the following requirements:

To fulfill the first requirement, we should be able to support private peer groups. This means also that we need to be able to run an own JXTA infrastructure without using foreign resources on the net. This is exactly what the JXTA specification recommends to do for peer applications. To enable peer discovery in JXTA (second requirement), we need a set of services: Obviously, we need at least the *discovery service* which allows us to publish and query for (peer) advertisements. Then, we need the *resolver service* which the discovery depends on. But to get into contact with unknown peers, we heavily rely on the *rendezvous service*. The rendezvous server is the first peer we contact and the advantages of the discovery service only is possible with the help of the rendezvous. A rendezvous server is needed for the entire peer group, not every peer needs to implement the server part. An Origo peer may be a rendezvous client only. Therefore, we concentrate on the client part and note the possibility to run the rendezvous server as a JXTA JSE peer. As all discovery messages (and also messages from the resolver and rendezvous) are based on normal JXTA messages, we clearly need the *endpoint service* together with a transport module. Having these, we honor also the third requirement of providing a message transport.

There is still a missing service: the *endpoint router*. Peers are addressed with peer IDs, so we need the endpoint router to resolve the IDs to addresses that specify the transport protocol and the exact address. This is only a

small task of the router. There is no urgent need for the other functionality enabling us to have peers behind firewalls and NAT gateways.

Summing up the set of required services, we get the list shown in table 4.2. The requirements are fairly vague but we will provide more detail later in this Section and show the resulting challenges when presenting the implementation in Chapter 4.7.

Service module	*Functionality*
Endpoint service	Message layer abstraction
A transport module (e.g. TCP)	Message sending and receiving over the wire
Endpoint router (parts)	Routing messages to available gateways selecting a fast transport
Rendezvous service (client)	Connection to peer group
Resolver service	Query-response system
Discovery service	Advertisement querying and publishing

Table 4.2: Required services

We now take a closer look at the design of VamPeer. By first introducing the module structure, we see how the entire platform works.

4.6.1 Module structure

The JXTA structure is very modular; every service and every peer group, even the platform (the world peer group) itself, is a module.

A module is an entity that can be started, suspended and stopped. This enables the VamPeer platform to perform the entire start up process without knowing every internal detail of every module.

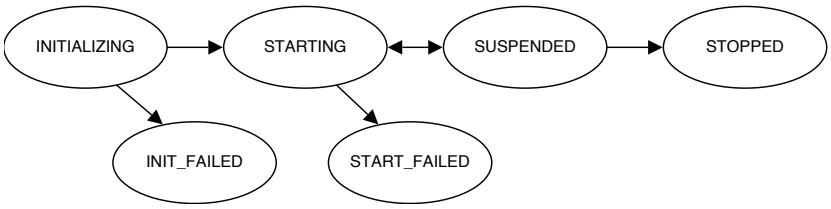


Figure 4.1: Module life cycle

Figure 4.1 shows a life cycle of a module. After successful loading where usually the basic initialization like creating data structures is done, a module

can be started. The `suspended` mode is available to temporarily stop a service in order to make it rest for a while in standby. The `start` method has to take care of the two possible calling states. To permanently shutdown a module, one can call `stop` in suspended mode. A stopped module should not and cannot be started again. If you really need to do this, you have to create a new module instance.

There are also some states indicating fatal errors. When one of them has occurred, a module should not be touched again. Only the constructive operations `init` and `start` are allowed to produce errors. `suspend` and `stop` are always expected to function properly.

A module is represented in VamPeer with the deferred class `P2P_MODULE`. Figure 4.2 gives an idea about the classes which effect it. Also, we already see how the peer groups are related with modules; that is what we will look at in the next Section.

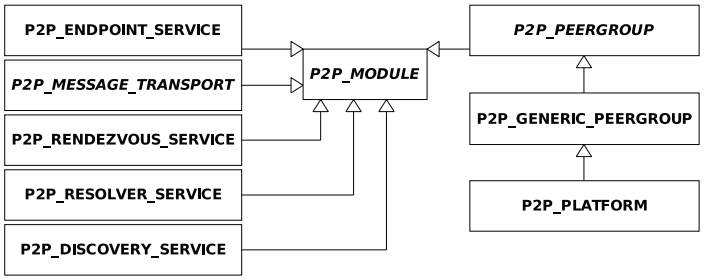


Figure 4.2: Module class hierarchy

4.6.2 Peer group modules

As a peer group specifies the available services for its group, it makes sense to make a peer group responsible for managing its services. So we just have to start the peer group when we would like to start the application's P2P support with all services. Therefore, the peer group is also a module which can be started and stopped (`P2P_PEERGROUP`).

Unfortunately, it is not that simple to start the entire P2P platform. To load a module we need a parent peer group, an ID and a module implementation advertisement used for configuration. Thus, we need a bootstrapping process that handles the loading of our main peer group – usually the net peer group.

This is exactly the purpose of our world peer group (`P2P_PLATFORM`). In VamPeer, it is not used for anything else but loading the NPG. As the WPG is itself a peer group, it is also a module but with a different creation procedure. This allows us to retrieve the needed data like the configuration directory path and the logger object.

It is not yet clear enough why we do not just adapt the net peer group to manage platform creation. At the moment, only one real peer group can be run in VamPeer.

4.6.3 Defining a peer group

Defining a peer group implies to prepare several requirements:

As a peer group is in the first place a normal module, we first have to establish the module configuration. This requires to have a module class ID (MCID) and a module specification ID (MSID) which identify the local and remote behavior as already pointed out in Section 4.4.

Another part of the group definition is built by the group ID, a name and a description. Together with the MSID, we are now able to build the peer group advertisement.

Modules are loaded by the MSID. This means that the module loader gets a MSID (besides an arbitrary ID and a name) in order to load the correct module code. Hence when loading a peer group, we have to provide its MSID.

As peer groups define which services (modules) they provide, the peer group module code is responsible to load these modules. The group module therefore somehow contains a list of needed module MSIDs. In our standard group implementation `P2P_GENERIC_PEERGROUP`, this list is called `modules` and `define_modules` is the method that initializes it. As we use a parent-child relationship between groups, the group services are usually inherited by a child group but one may easily redefine them.

When we inherit group services, we share the module instance. This clearly makes sense for certain services which are not group context sensitive. The top group which defines such a service is responsible for it (it is the only authorized group to load, start and stop this module). To respect this rule, the module list of the group also has to keep track of whether a module is inherited or newly defined.

When a module is loaded, the loader and later also the module itself should be able to access the module implementation advertisement. The loader may need the advertisement to know what module code to load for the given MSID. The module itself may use the advertisement to lookup some configuration parameters. Thus, the module implementation advertisement has to be available for each used module.

Conventionally, the group module implementation advertisement contains all advertisement of its modules. The peer group module will then extract these and make them available.

There is one open issue with module loading: a running module has itself an ID. We speak thereof of the *assigned ID* because the module loader assigns an arbitrary ID to the module. Usually, we assign the MCID but it is not necessary to do so. The module uses its ID to create a unique handler name when registering with other services.

Summarizing, we need an implementation advertisement for each module, where the one for the group contains all advertisements for its services. Additionally, we should provide a peer group advertisement to declare the group module as a JXTA peer group.

The entire module loading procedure is quite generic because of the dynamic module loading. Although Eiffel does not provide this, we choose to stick to the convention and provide also these implementation advertisements even if we only may use it to parametrize modules.

4.6.4 Services

With the strict module structure, we are basically done with presenting the VamPeer's design because everything is bundled into a module and therefore every service looks quite similar. Nevertheless, we have to describe some particularities, especially how services interact among each other.

4.6.5 Module choice

Each module resides in its own Eiffel cluster together with its related classes, namely the XML document types. Module unspecific classes are located in the main `vampeer` cluster. See Figure 4.3 for a clusters overview.

Listing 4.3: Clusters overview

```
vampeer/ (38 classes)
|-- discovery/ ( 3 classes)
|-- endpoint/ ( 4 classes)
|-- pipe/ ( 2 classes)
```

```
|-- rendezvous/ ( 5 classes)
|-- resolver/ ( 5 classes)
`-- transports/ ( 7 classes)
   |-- router/ ( 4 classes)
   `-- tcp/ ( 5 classes)
```

The pipe service[15] is not implemented but the module specification advertisement depends on the pipe advertisement.

Until now, we have spoken of generic transport modules but finally we implement only one, the TCP transport.

TCP is actually a misleading name as most of the other transports indirectly are TCP based as well. But with the TCP transport, we directly reside on top of TCP using the JXTA wire representation for messages. It is the most simple but also the fastest transport.

4.6.6 Service layers

We would like to clarify how all the JXTA services are related to each other. Particularly, how messages are passed through. We do so by looking at two examples which cover all services implemented in VamPeer. We first treat an outgoing and later an incoming message.

Outgoing message: A discovery query which is sent out to be propagated in the group. It can be a general query to find new peers. Figure 4.3 shows the UML sequence diagram hiding the exact operation signatures. Below, we comment each method call:

1 When the discovery service is called with `remote_query_advertisements`, it creates a `P2P_RESOLVER_QUERY` containing the discovery query string and a handler name (see the incoming message example for its use). It then passes it to the resolver which is requested to propagate the query (instead of just sending the message to a single peer).

2 The resolver service then creates a `P2P_ENDPOINT_MESSAGE` with a message element containing the resolver query string. It passes the message together with the resolver service name to the rendezvous service.

3a The rendezvous adds another message element with some metadata to the message. This informs the recipients that the message was propagated (important when they do repropagation). The rendezvous then

[15]The *pipe service* implements the pipe binding protocol and provides virtual communication channels among several peers.

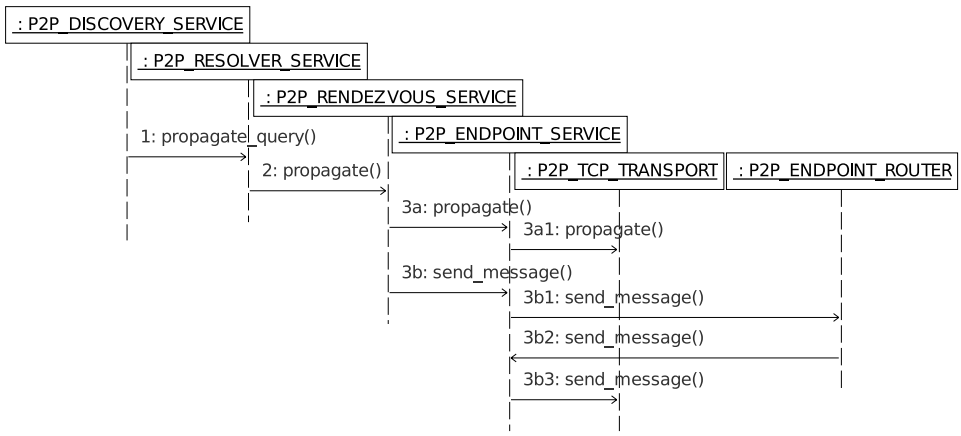


Figure 4.3: Information flow for an outgoing discovery query

first propagates the message in the local network by simply calling the endpoint propagation service.

3a1 The endpoint service now passes the message and the service name to each transport propagation method. Actually, it makes only sense for the TCP transport, as it supports IP multicast (other transports just ignore the call). However, multicast is discouraged[16] and mostly turned off in the platform configuration.

3b When the peer is connected to a rendezvous server, it is able to do propagation via this server. So the rendezvous service sends the message to the peer ID of the server by calling the endpoint service `send_message`. For this, it has to create a `P2P_ENDPOINT_ADDRESS` with the `jxta` protocol and the peer ID of the server[17].

3b1 To resolve the endpoint address, the endpoint service passes the message to the endpoint router which is the registered transport module for the `jxta` protocol.

3b2 The endpoint router does a (local) lookup for the given destination peer ID (querying for peer and route advertisements). As soon as a gateway

[16]Multicasting should not be used because it causes much network traffic and may stress some smaller edge peers. It poses also a risk for developers that test in local networks only because things may work locally with multicast but maybe will not with remote peers.

[17]E.g. `jxta://rdv-peer-id/rsv-service-name` (while rsv-service-name is the resolver service module class ID).

and a transport protocol is chosen, it calls the endpoint service again to deliver the message with a rewritten, specific destination address[18].

3b3 The endpoint service now passes the message to the specified transport protocol which tries to connect to the specified peer and writes the message to the wire. Note that there is neither a feedback to the caller whether the message dispatching has been successful or not, nor an acknowledgment message from the other peer. The TCP transport is therefore seen as an unreliable transport.

Incoming message: The path for incoming messages is somewhat shorter. We will continue our example and let the peer receive a discovery response. To explain the three method calls, we first look at Figure 4.4 which presents the calling sequence.

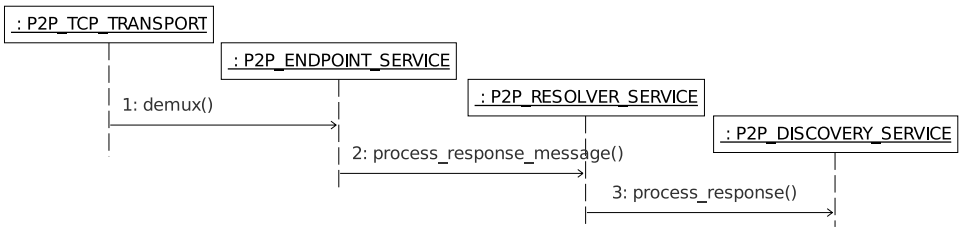


Figure 4.4: Information flow for an incoming discovery response

1 As soon as the message transport has received a message and the message parsing from the wire has been successful, it calls the `demux` method of the endpoint service with a `P2P_ENDPOINT_MESSAGE` instance.

2 From the delivered message, the endpoint service extracts the destination address and from that the service name. Then it calls the registered agent for this server name which is owned in our case by the resolver service.

[18] E.g. tcp://129.132.105.170:9700/rsv-service-name

3 The resolver interprets the resolver message element and is able to extract the response string and a handler name. It then calls the registered agent for this handler name and passes a `P2P_RESOLVER_RESPONSE` object.

The discovery service will finally parse the resolver response string and create a `P2P_DISCOVERY_RESPONSE`. From there, the delivered advertisements (responses) are either published locally or they may be passed to a further agent registered by a user service.

4.6.7 Address rewriting

To stay compatible with other JXTA implementations, we have to pay attention to a special topic: *address rewriting* (address mangling). The problem is that messages may not be directed to the correct service in other implementations since they may have a slightly other peer group hierarchy.

In a JXTA platform, only one module can actually handle network traffic (one module per transport protocol). But generally, it is possible to run multiple peer groups at a peer such that each group has its own endpoint service. Hence, the question is where we register the transport module(s).

As the JXTA protocol does not specify this, each implementation can do as it likes. While in JXTA JSE the WPG owns the transport modules, JXTA-C and VamPeer settle them in the NPG. The reason for that is that in Java, the WPG is a real peer group whereas JXTA-C does not have a WPG and we in VamPeer only use it for platform configuration purposes.

With multiple endpoint services and in order to receive messages, services have also to be registered in the top peer group that owns the transport modules. Such an indirect registration, which is automatically done behind the scenes, is done with a *mangled* address which includes the original peer group ID where the service is registered in the first point. Figure 4.5 shows how a mangled address is assembled.

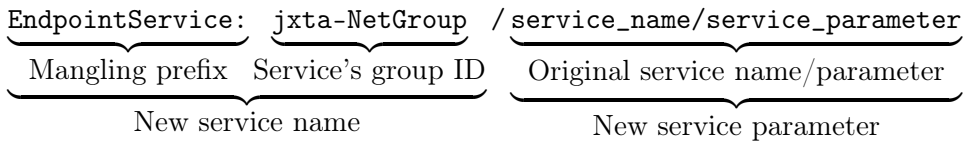


Figure 4.5: A mangled service handler name

With such a mangling scheme, it is clear that also the message destination address has to comply to this rule. To send a NPG discovery message to a Java peer, we have to mangle the address before because the discovery is in the NPG but the Java transports are in the WPG. Sending the same message

to a VamPeer peer, we would not have to mangle the address because its transport modules are also in the NPG.

In the reverse direction, a Java peer will send discovery messages always mangled and we would discard the message because we have no mangled address registered for it as our discovery service already runs in the top group NPG.

To overcome this, we register our services always with the mangled address additionally to not loose any messages from Java peers. And to send messages, we always mangle the destination addresses, but we do not force it; hence, it is still possible to speak with other VamPeer peers without group mangling.

4.6.8 Rendezvous propagation

In P2P networks, propagation is a central service because for many messages we do not know which peer exactly uses the information. So we propagate the message to everyone and hope that some peers use or process it. As we have seen, propagation with group scope (instead of local network only) is done relying on the rendezvous service.

When we hear about message propagation, we are tempted to classify it as flooding the network. While this can be a solution (previous versions of JXTA JSE did this), it is too traffic intensive and extremely inefficient.

We already mentioned in Section 4.4, that the rendezvous passes the messages to the appropriate services which decide, based on some gathered knowledge, if the message should be repropagated or not.

This means that we have to get away from the idea that we can only place a JSE rendezvous somewhere and propagation just works. Either, we design our peer application to use only standard services or we implement the user services also in Java on the rendezvous peer. The third and best solution would be of course to have an Eiffel rendezvous implementation.

4.7 Implementation

In this Section, we look at the implementation of VamPeer and describe solutions for the most important tasks. The use of the library is documented in Section 4.10.

4.7.1 Dependencies

VamPeer is using several software libraries. The *Gobo* [76] library is used for data structures, date/time and its XML generation and parsing support.

As we are mainly dealing with remote peers, we need a networking library. We looked for a simple solution and got to the *EiffelNet* code [58]. Unfortunately, the library is not used by many people, resulting in an outdated API which is not always useful for each task. Therefore, we wrote an extension, described in Section 4.7.2.

4.7.2 Socket extensions

The main problem with *EiffelNet* is the missing support for timeouts. Whenever we wait for network data, we are not able to wait eternally. Generally, there are two kinds of interrupts in these situations in which we like to quit the reading/waiting task: first, when a certain time has been passed and second, when we get an internal request for closing the connection such when the user application is shutting down.

Another lack of *EiffelNet* is buffering. Usually, we want to assemble a network message and send it as *one* packet. The straightforward idea to just use a string does not really work well because with binary data, we have to append 8-, 16- or 32-bit integers (respecting the network byte order!) – this results in unmaintainable code. Additionally, we sometimes have to know how large a buffer is (for example to specify a message body length).

We therefore wrote a helper class `P2P_SOCKET_EXTENSIONS` which provides exactly these features: timing and buffering. We did not choose to create a child socket class because the helper methods may be used for several types of sockets: TCP or UDP. We provide methods for writing by using a buffer, for reading and for some socket checks.

One may fill the buffer with strings and integers (from 8 to 64 bits, using big-endian format). The buffer is a string and may be adapted and used at will. As soon as the buffer is sent, the buffer is emptied again.

For the read methods (returning a string or the various integer types), one has to set a timeout first. They read and wait until either they got exactly the expected amount of data, they pass the timeout or a given constraint has become active. The last two cases raise an exception. Of course for strings, there is also a method reading *up to* a given data length (in many cases, we do not know exactly how much data we are expecting).

The socket checks provide a way to wait until a connect request is successful within the timeout and a way to check generally whether data is available for reading.

4.7.3 XML documents

JXTA uses XML documents. Every advertisement is in XML, and also every higher level JXTA message. Therefore, we need a simple way to parse and generate XML documents.

Figure 4.6 shows our class structure for XML documents. We hide the exact operation signatures for simplicity and do not list all descendant classes.

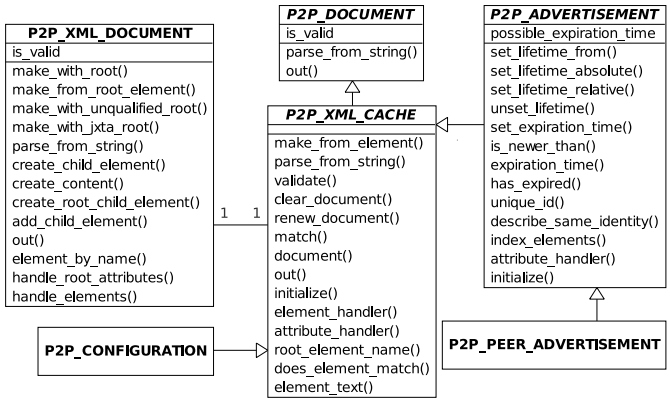


Figure 4.6: XML document class hierarchy

The class `P2P_XML_DOCUMENT` manages most handling using the *Gobo XML* interface. It is not an abstraction so that the underlying XML library can easily be replaced; it rather provides helper methods just to simply build an XML tree from elements, to get the content as a string and to build a tree parsed out of a string. When parsing, we let Gobo build the full tree and we afterwards provide callbacks for each root element and root attribute.

Its main client class is `P2P_XML_CACHE` (deferred) where all XML document classes inherit from. It mainly declares central document methods, such as creation, validation, output and element matching. It inherits from `P2P_DOCUMENT` which provides an interface for very generic documents. The XML tree is not built until `out` is called the first time. Further calls return a cached XML string unless any element has been changed (`renew_document` should always be called internally in element setters).

To create a new XML document type, we just need to inherit from `P2P_XML_CACHE`, define some setters/getters for the actual content elements and implement `match`, `root_element_name`, `attribute_handler`, `element_handler` (to gather data after the parsing process) and `document` (to create the XML elements). We may also redefine `initialize` and `validate`

and add additional creation methods.

This interface is appropriate for general XML documents, we require an additional interface for advertisements: `P2P_ADVERTISEMENT`. Each advertisement has a unique ID used for the advertisements store in the discovery service. It should also define its lifetime and the remote expiration time. For SRDI, an advertisement should also define some elements which can be indexed.

Hence, we declare a unified interface for advertisements as shown in Figure 4.6. All advertisement classes (such as `P2P_PEER_ADVERTISEMENT`) should effect it.

4.7.4 Using UUID for JXTA IDs

For VamPeer, we use JXTA IDs in the UUID format as it is the case in JXTA JSE. Look at Figure 4.7 to get an overview of the ID class hierarchy.

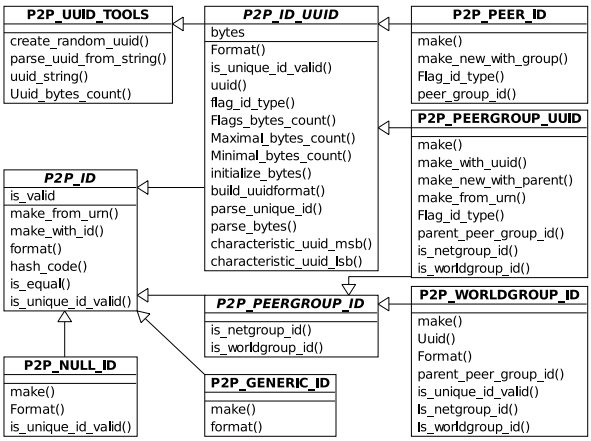


Figure 4.7: ID class hierarchy

`P2P_ID` is the main interface for an ID. Such an ID can be parsed and written as a URN string (it is comparable and hashable). Direct descendants can be instantiated are `P2P_NULL_ID` (for referencing no resource, actually never used in VamPeer) and `P2P_GENERIC_ID` which may contain any valid ID in any format.

We support the `uuid` format using the deferred class `P2P_ID_UUID`. It parses the unique ID part and creates a byte array. For each ID type (such as peer, peer group, codat ID...), there is an effecting class defining the byte interpretation. Figure 4.7 shows two of them. Each such type also defines

the characteristic UUID bytes, made available through the feature `uuid`[19].

`P2P_UUID_TOOLS` is actually not related to the ID class structure but provides helper methods to create a random UUID and to parse/write a UUID. In VamPeer, UUIDs are everywhere seen as `ARRAY [NATURAL_8]` with a capacity of 16 bytes.

For peer group IDs, there is a special interface `P2P_PEERGROUP_ID`. It is independent on the `uuid` since we use other formats as well for peer group IDs, for example the `jxta` format for the net and the world peer group. The `uuid` version `P2P_PEERGROUP_UUID` is therefore descendant of two classes, `P2P_PEERGROUP_ID` and `P2P_ID_UUID`.

It may be confusing that the `jxta` peer group IDs also provide a `uuid` feature. It is used to generate child peer group IDs because children point to the parent group uuid. The following examples try to clarify these ID relationships:

World peer group ID: `urn:jxta:jxta-WorldGroup`
Implicit UUID: `596162616461626l4A78746150325033`[20]

Private net peer group ID:
`urn:jxta:uuid-`$\underbrace{\texttt{822A7C9E6B804759870B81B10070E9C9}}_{\text{Group UUID}}\underbrace{\texttt{596162616461626l4A78746150325033}}_{\text{WPG's characteristic UUID}}\texttt{02}$

Peer ID, member of the private net peer group:
`urn:jxta:uuid-`$\underbrace{\texttt{822A7C9E6B804759870B81B10070E9C9}}_{\text{Group's characteristic UUID}}\underbrace{\texttt{2F3F01367359485B95D5C6CFA82B9775}}_{\text{Peer UUID}}\texttt{03}$

4.7.5 Threads

We address this Section to thread management in VamPeer, because we want to present which threads exist and why users have to pay attention to some details.

Threads are an absolute must when dealing with the network. But threads are also a source for many software bugs because we cannot rely on all contracts anymore in a multithreaded environment. Other threads may change something in course of time. This is why we have to use locking mechanisms to ensure exclusive access for a certain amount of time.

[19] While the peer group UUID in a peer ID is just meta information, the peer UUID is the characteristic part.

[20] In ASCII, the 16 bytes stand for "YabadabaJxtaP2P3". Actually, the last byte was an error; it should be a "!" but they looked up the wrong number in the ASCII table (decimal instead of hex)...

While we have `synchronized` environments in Java, Eiffel currently does not support such a mechanism in the language itself[21]. We therefore have to use the possibilities from *EiffelThread* which provides read/write locks (for multiple readers, one writer), mutexes (simple locking) and condition variables (for thread synchronization).

In VamPeer, we use at least four threads besides the main thread. We will describe them now:

Main thread: First, we will look at what the main thread is used for. It is the thread which starts the platform and is fully controlled by the user application. Thus, the entire platform startup is done and all the other threads are created by the main thread. Since VamPeer does not provide an event loop, the user application has to handle its main thread itself. The main thread is also used to shutdown the platform.

Most of the other threads are created by the TCP transport and its related classes:

Server thread: The first thread created is the server thread. `P2P_TCP_SERVER` binds a server port and listens for incoming connections. When a new connection is accepted, it passes the socket to the main TCP class (`P2P_TCP_TRANSPORT`) which handles the connection with other threads. The server thread can be closed by calling `shutdown` (from another thread). The socket will then be cleaned up and cause the server thread to terminate instantly.

Connection threads: Each TCP connection gets its own thread. As soon as a socket is passed from an incoming connection, or when a new socket has to be created, the connection object `P2P_TCP_CONNECTION` launches a new thread.

The thread manages waiting during connection setup and waiting for incoming messages via its socket. It is important to know that message processing (for received messages only) is handled by the connection thread. This means, a service should never do extensive work in a processing agent.

It is therefore not possible for a message handler to wait for another message retrieved with the same connection. However, sending messages is allowed in such a handler (see next paragraph).

A connection thread terminates itself after a certain time but can of course also be destroyed by calling `close`.

[21] SCOOP will be a good solution, but it is still a research subject.

Message queue thread: Because services should not be affected by the slow message delivery, outgoing messages are queued. The calling service thread returns instantly and without result as soon as the message has been successfully stored in our message queue.

`P2P_TCP_TRANSPORT` manages a message queue thread which waits for and gets triggered on arrival of new messages. It then looks up a possible existing connection or tries to open a new one. As soon as a connection state is in a valid mode, it sends the message and removes it from the queue. It tries several times when dispatching fails. Summarizing, we have two fix threads for the TCP transport plus one thread per connection. In suspend mode, all connections are closed and the server- as well as the queue manager thread are stopped.

Rendezvous thread: The rendezvous service owns another thread. It is used to maintain contact with rendezvous servers meaning to renew connection leases. Hence, this thread is mostly sleeping.

The thread is launched when the service is started and stopped when the service gets suspended. During start, we have a bootstrapping problem because not all services are started yet and the system might only be partially functional. We cannot start sending messages in this state and have to wait until the platform is started entirely.

That is the reason why the peer group is offering a method `when_fully_started` returning `True` as soon as all group services are started successfully or `False` on a failure. The rendezvous connection manager thread calls this method before it begins to contact a rendezvous server to get a lease.

Discovery SRDI thread: The discovery service creates a thread to manage SRDI pushes. It regularly looks for new local advertisements and propagates its index changes. Like the rendezvous connection manager, it waits with index pushing until the platform has been started fully. The thread is stopped also in suspend mode.

4.8 Advertisement store

Services and user applications should not have to store advertisements themselves but should be able to access a central store containing all advertisements of the system. This store is implemented by the discovery service with its local query methods. We would like to describe its advertisement store here. The store can be divided into two parts, persistent and memory store.

4.8.1 Persistent store

When starting the platform, a configuration directory must be declared. This is used to store platform configuration and advertisements.

Listing 4.4: Persistent store directory layout

```
|-- Modules/
| |-- jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000010206.xml
| |-- jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000020106.xml
| |-- jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000030106.xml
| |-- jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000060106.xml
| |-- jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000080106.xml
| |-- jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000090106.xml
| `-- jxta:uuid-DEADBEEFDEAFBABAFEEDBABE0000000B0106.xml
|-- jxta:jxta-NetGroup/
| |-- Advs/
| |-- Peers/
| | `--jxta:uuid-59616261646162614E504...3D4C31803.xml
| `-- PeerGroup.xml
|-- jxta:jxta-WorldGroup/
| |-- Advs/
| |-- Peers/
| | `--jxta:uuid-59616261646162614E504...3D4C31803.xml
| `-- PeerGroup.xml
`-- PlatformConfig.xml
```

The file hierarchy layout is simple as shown in Listing 4.4. For each peer group a directory contains its group advertisement, one directory for general and one for the peer advertisements. Each advertisement is stored as a single XML file with shortened unique ID used as file name. For the module implementation advertisements, a special modules directory is maintained.

The persistent cache is only used when the platform is loaded. During initialization, configuration is read from disk or will be created and stored. When the configuration directory is not present, it is created.

All advertisements are loaded into memory once the discovery service is started. This enables peer applications to have advertisements permanently available.

The class `P2P_CACHE_MANAGER` is responsible for accessing and managing persistent cache. As soon as a platform instance is disposable, the cache manager is ready for access too.

4.8.2 LRU cache

The memory cache is implemented with a *Least Recently Used* (LRU) cache in the class `P2P_ADVERTISEMENTS_LRUCACHE`. It is inspired by the JXTA JXME project which is used for mobile applications with limited memory.

The LRU cache has a maximal capacity, so that the oldest, never used entries are discarded. Furthermore, expired advertisements are removed automatically.

The discovery protocol is designed to distinguish between peer, group and other types of advertisements. We have three LRU cache instances defined in the discovery service, one for each type.

Advertisements are accessed by their unique ID or by a key/value search which makes use of the advertisements `match` method. It is also possible to retrieve a number of random advertisements (or all of them).

Advertisements are identified by their unique IDs defined in the advertisement. When creating a new advertisement type, it is important to define a scheme for the unique ID. While it is not always possible to create a unique ID because the advertisement information is too common, we have to be aware that an advertisement cannot be stored in the cache without having a unique ID.

4.9 Shared creators

There are some important creators (also called factories) in VamPeer. All creators can be accessed using the class `P2P_CREATORS_SHARED`:

ID creator: The ID creator represented by `P2P_ID_CREATOR` provides globally unique instances for the null, NPG and WPG ID and provides the possibility to parse an ID string and to create the appropriate ID object out of it.

It knows and can handle the `jxta` and `uuid` format types. Custom creators may be registered. Such an agent gets a generic ID and has to return a specific ID object or `Void` if the ID is not recognized. The custom creators can override the VamPeer types.

XML document creator: Specific XML documents are also derived with a shared creator: `P2P_XML_DOCUMENT_CREATOR`. There are two methods to create XML documents. Both expect the XML root element.

The first method `document_from_element` returns a `P2P_XML_CACHE` object or `Void` whenever the creation was not successful. If the document type

cannot be detected, it returns a valid `P2P_UNKNOWN_XML_DOCUMENT`.

It is used by the second method `advertisement_from_element` which simply tries to cast the result to an advertisement (`P2P_ADVERTISEMENT`).

The document creator is also useful for user services as custom document types can be registered too, similar to the ID creator.

Wire message creator: The last creator is used for the endpoint message wire representation. Currently, there is only one mime type used (`application/x-jxta-msg`). Thus, the wire representation is always in binary format, handled with `P2P_WIRE_MESSAGE_BINARY`.

4.10 Using VamPeer

In this Section, we look at the API and present how VamPeer can and should be used. While we first list all possibilities, we describe examples which are supplied together with the library.

4.11 Platform starting

For starting and stopping the platform, we always deal with a `P2P_PLATFORM` instance. To get the instance, we call `make` passing the configuration directory path and a logger (`L4E_LOGGER`). The platform automatically reads the configuration file `PlatformConfiguration.xml`, if available, and sets the `is_configured` variable appropriately. If it is false, we have to provide a new `P2P_CONFIGURATION` as shown in Listing 4.5.

Listing 4.5: Configuring the platform instance

```
configure_platform is
    -- Configure VamPeer platform instance
  require
    Logger_valid: logger /= Void
  local
    conf: P2P_CONFIGURATION
  do
    create platform.make (".vampeer", logger)
    if not platform.is_configured then
      -- New NPG Peer
      conf := platform.default_configuration
      conf.set_name ("Peer name")
      conf.set_description ("Peer node description")
      platform.configure (conf)
    end
  ensure
    Platform_set: platform /= Void
end
```

The default configuration returns the settings for a new NPG peer: A new peer ID is created, the TCP settings are set to automatic interface/port detection and the rendezvous is set to client mode with the standard NPG rendezvous servers. Before the configuration is fed the platform, we set an appropriate peer name and a description.

The next step is to load the NPG. For the public NPG, this is easily done with the `standard_net_peergroup` method which takes care of module implementation advertisement creation and group loading.

Listing 4.6: Loadingthe platform with the public NPG

```
start_platform is
    -- Load and start net peer group
  require
    Platform_configured:
      platform /= Void and platform.is_configured
  do
     -- Load NPG
    npg := platform.standard_net_peergroup
    if
      platform.module_status /= platform.init_failed
    then
       -- Start Platform/NPG
      platform.start
    end
    if
      platform.module_status /= platform.start_ok
    then
      npg := Void
    end
  ensure
    Npg_set: npg /=
      Void implies npg.module_status = npg.start_ok
  end
```

Modules can make use of start parameters which are provided with the `start_with_arguments` command (expecting an `ARRAY [STRING]`). All started modules receive the same arguments. However, the JXTA services do not use them.

When the platform is successfully started, we can access the services via the NPG peer group instance. Note that at this point, the rendezvous client may not yet have connected to a server and propagated messages might get lost.

We should not forget to maintain an event loop when we are done with initializing, or the application will quit instantly. The platform does not provide such a feature.

To stop the platform again, we call `platform.stop`. If you just like to pause the P2P activity, you will like the `suspend` command (see Section 4.6.1).

4.11.1 Private peer groups

It is somewhat more complicated to load a private peer group since we have to provide and create more specific settings. The entire process looks very

similar, but we have to create another configuration and load the private NPG differently.

Creating new IDs: The very first step is to create all the IDs used for the new peer group. Please read the introduction in Section 4.6.3 at page 58 to get an overview about the definition of a peer group.

Listing 4.7 shows how to generate the three IDs used for a new private NPG: a MCID and a MSID for the peer group module and finally the peer group ID. A private NPG is always a child of the WPG, so we use the WPG UUID as parent ID.

Listing 4.7: Creation of IDs for a new peer group

```
create_peergroup_ids is
    -- Create all IDs used for a new peer group
  local
    mcid: P2P_MODULE_CLASS_ID
    msid: P2P_MODULE_SPECIFICATION_ID
    gid: P2P_PEERGROUP_ID
    wpgid: P2P_WORLDGROUP_ID
  do
    create mcid.make_new
    create msid.make_new_with_class (mcid.uuid)
    create wpgid.make
    create gid.make_new_with_parent (wpgid.uuid)
  end
```

In a P2P application the code above cannot be used; the creation of the new IDs is a one-time process. One has to hard code the new IDs for an application. Instead of using the example above, the *idcreator* example shows how to generate all the different kinds of IDs.

Creating a configuration: To create an appropriate configuration, we have to set the peer ID, name and description in a `P2P_CONFIGURATION` object as shown in Listing 4.8. Additionally, we have to add service configurations for the TCP and the rendezvous module. For the TCP configuration, one may choose the default with `default_tcp_configuration`. The rendezvous configuration needs to be adapted to the groups rendezvous servers, see Section 4.12.3 for more detail.

Listing 4.8: Creating a platform configuration for a private peer group

```
new_configuration: P2P_CONFIGURATION is
    -- New configuration for a private peer group
  require
    Group_id_valid: gid /= Void and gid.is_valid
  local
    pid: P2P_PEER_ID
    rdvconf: P2P_RENDEZVOUS_CONFIGURATION
  do
     -- New peer ID
    create pid.make_new_with_group (gid.uuid)
     -- Rendezvous client configuration
    create rdvconf.make
    rdvconf.add_seed_uri ("http://stablehost.org/rdvs.cgi"
       )
    create Result.make_with_id (pid)
    Result.add_service_parameter
      (transport_tcp_mcid, default_tcp_configuration)
    Result.add_service_parameter
      (rendezvous_mcid, rdvconf)
  ensure
    Result_set: Result /= Void and Result.is_valid
  end
```

The constants for the TCP and rendezvous MCIDs are listed in `P2P_CONSTANTS`.

Creating a module implementation advertisement: A module implementation advertisement for the group should be created when it is not available in the cache yet. Listing 4.9 shows how we first can get a standard advertisement containing all advertisements for the JXTA services. We then add a user service and store the entire document to disk using the cache manager.

Listing 4.9: Creating a peer group module implementation advertisement

```
set_peergroup_implementation_advertisement is
    -- Make sure that peer group
    -- implementation advertisement exists
  require
    Platform_valid: platform /= Void
  local
    params_doc: P2P_XML_DOCUMENT
    pg_mia, smia: P2P_MODULE_IMPLEMENTATION_ADVERTISEMENT
  do
    if
      not platform.cache_manager.
        has_module_implementation_advertisement (pg_msid)
    then
      pg_mia := platform.
        peergroup_implementation_advertisement
        (pg_msid, "PG_CLASS", "group description")
      params_doc := Result.parameter.document
      -- Add user service impl adv
      smia := platform.
        default_implementation_advertisement
        (service_msid, "SERVICE_CLASS",
        "service description")
      params_doc.
        create_root_child_element ("Svc", namespace_empty)
      params_doc.
        add_child_element
        (params_doc.last_element,
        smia.document.document.root_element)
      -- Store group impl adv
      platform.
       cache_manager.
       store_module_implementation_advertisement (pg_mia)
    end
  end
```

Loading a private NPG: To load a private peer group, the platform provides the method `load_net_peergroup` which expects the group ID, the specification ID and an agent for instantiating the group.

We do not like to load the peer group module directly because we want to have a unified access through the platform object. This also allows the platform to be up to date with its module status.

Listing 4.10: Loading a private NPG

```
load_peergroup is
    -- Load a private net peer group
  do
    npg ?= platform.
      load_net_peergroup (gid,
      pg_msid, agent peergroup_loader)
    if
      npg /= Void and npg.
      module_status /= npg.init_failed
    then
      npg.group_advertisement.
        set_name ("Group name")
      npg.group_advertisement.
        set_description ("group description")
      platform.cache_manager.
        store_peergroup_advertisement
        (npg.group_advertisement)
    end
  end

peergroup_loader (a_pg: P2P_PEERGROUP;
  an_id: P2P_ID;
  a_mia: P2P_MODULE_IMPLEMENTATION_ADVERTISEMENT):
  P2P_MODULE is
    -- Private NPG loader
  do
    if
      a_pg = platform and an_id.
        is_equal (gid) and
        pg_msid.
        is_equal (a_mia.specification_id)
      then
      create {PG_CLASS} Result.
        init (a_pg, an_id, a_mia)
    end
  end
```

4.12 Using Services

After starting the platform, we will only work with the peer group instance and its services. All modules are registered in the peer group, so we may access them through our NPG. While the JXTA services have an easy access method (for example `npg.endpoint_service`), we have to access user services through their module name: `npg.lookup_module("servicename")`.

4.12.1 Endpoint service

The endpoint service is central, because all other services directly or indirectly rely on it. This includes all services and as well as all transport modules.

Receiving endpoint messages: We first look how services use the endpoint service. When they are ready to receive endpoint messages, they register a service name, an optional service parameter and a handler using `extend_service`. The service name is the assigned module ID of the registering service and is therefore unique.

The actual, internal handler name is the combination of the service name and the parameter. When we specify a parameter, we get only those messages that exactly match the service name *and* the parameter. If no such handler exists, the handler matching only the service name will be called. Note that we can only have one handler per service; when a handler is registered, a possibly old handler for the same service name/parameter is silently replaced.

Message handlers get an endpoint message together with the extracted source and destination endpoint address. Handlers should not do long processing jobs and should return as soon as possible.

See Listing 4.11 for an example message handler. It prints out the content for the message element with name "dummy" from the user namespace.

Reaching remote peers: For actively reaching other peers, there are three possibilities:

1. `send_message` and `send_message_mangled`
2. `propagate` and `propagate_mangled`
3. `ping`

To send a message, we call one of the send message methods passing an endpoint message together with an endpoint address. The endpoint address

Listing 4.11: Example endpoint message handler

```
process_message
  (a_msg: P2P_MESSAGE; a_source,
  a_destination: P2P_ENDPOINT_ADDRESS) is
    -- Process incoming endpoint message
  require
    Message_valid: a_msg /= Void
    Source_valid: a_source /= Void
    Destination_valid:
      a_destination /= Void and a_destination.
         service_name.is_equal (sname)
  local
    msgel: P2P_MESSAGE_ELEMENT
  do
    msgel := a_msg.
      element_by_namespace_and_name
      (a_msg.namespace_user, "dummy")
    if msgel /= Void then
      print (msgel.content)
    end
  end
```

Listing 4.12: Creating and sending an endpoint message

```
send_endpoint_message (a_dest: P2P_PEER_ID) is
    -- Send endpoint message to 'a_dest'
  require
    Dest_valid:
      a_dest /= Void and a_dest.is_valid
  local
    ea: P2P_ENDPOINT_ADDRESS
    msgel: P2P_MESSAGE_ELEMENT
    msg: P2P_MESSAGE
  do
    create ea.
      make_with_id (a_dest, "pingservice", Void)
    create msg.make
    create msgel.
      make_string (msg.namespace_user,
      "Data", Void, "Ping")
    msg.extend (msgel)
    peer_group.endpoint_service.
      send_message (ea, msg)
  end
```

should use the protocol `jxta` with a peer ID. Listing 4.12 shows an example for this.

To learn if you need the "mangled" version or not, you may revert to Section 4.6.7 on page 63. However, using `send_message_mangled` is usually the safer way, but requires to pass the group ID of the calling service.

The propagate methods pass the message to all transports which should make use of transport specific propagation techniques. Currently, only the TCP transport can handle this request by sending a UDP multicast packet[22]. However, our TCP module implements only outgoing multicast and cannot read any incoming multicast messages.

Using the endpoint propagation method will at best reach peers in the local network, never the entire peer group. To propagate a message, we do not pass a full destination address because it is protocol and destination unspecific; we just pass the destination service name and parameter.

While the methods for sending or propagating messages are normally asynchronous, the ping command is a time-consuming call as it waits for a remote answer. We do not send messages with `ping`, we just use it to check

[22]The message size is therefore limited to 16KB.

if the given endpoint address is valid and available to us or not. The TCP transport for example tries to open a connection and to do a handshake with the remote peer.

Transport handling: Transport modules register with the endpoint service but use the method `extend_message_transport`. Transports may either be responsible for incoming (`P2P_MESSAGE_RECEIVER_TRANSPORT`) or outgoing messages (`P2P_MESSAGE_SENDER_TRANSPORT`) or both; the endpoint service is able to deal with these types.

Transports may induce messages by calling the `demux` command which analyzes the message and passes it to the appropriate service. The endpoint service will simply ignore messages for which no handler exists.

Message filtering: The endpoint service can filter messages. It is possible to extend filters for incoming or outgoing messages. Filters may not only decide whether a message is discarded or not but may also alter the messages. Listing 4.13 shows how such a filter handler can look like.

Listing 4.13: Example endpoint message filter

```
message_filter
  (a_msg: P2P_MESSAGE;
  a_src, a_dest: P2P_ENDPOINT_ADDRESS):
  P2P_MESSAGE is
    -- Discard incoming messages, if 'ignore_all' is set
  require
    Message_valid: a_msg /= Void
    Source_valid: a_src /= Void
    Destination_valid: a_dest /= Void
  do
    if ignore_all then
      logger.info ("Discarding message from: "
      + a_src.out)
    else
       -- feed message back to other filters and services
       Result := a_msg
    end
  ensure
    Result_set: ignore_all = (Result = Void)
  end
```

4.12.2 TCP this is the last candidate. next esc will revert to uncompleted text. ransport module

For the TCP transport module, we will not show the interface, because a user never gets into direct contact with the module features. But we want to show how a user can configure the transport.

The TCP transport looks for its service parameter in the platform configuration. The configuration must exist or the module will not start. There are currently three values to set: the `Port`, the `InterfaceAddress` and the `MulticastOff` flag.

Listing 4.14: Full TCP transport configuration

```
<Parm type="jxta:TCPTransportConfiguration">
  <MulticastOff></MulticastOff>
  <Port>9701</Port>
  <InterfaceAddress>
        129.132.105.170
  </InterfaceAddress>
</Parm>
```

When the *MulticastOff* flag is set, the TCP module will disable propagation via multicast. When the flag is not set, only sending of multicast messages is supported. The module currently does not support listening for multicast messages. We should therefore disable multicast.

The *port* specifies the server port. If it is not set, the module will automatically choose a port in the range 1024–65'535.

The *interface address* also concerns the TCP server. When the address is set, the module will only accept messages for the given address. We may use this in combination with VPN (virtual private network) to lock out messages from unauthorized sources.

When the interface address is not specified, the module tries to detect the interface and only listens on this interface. It is important for the platform to know its own IP, namely to create the peer advertisement which includes the route information.

The platform can not always detect its IP correctly. It has to use a connection to find out the local IP. This is done when the rendezvous seed URL is resolved when the platform is started (see next Section). When the platform does not need to get the rendezvous seed list, it will set the local IP to `127.0.0.1`.

4.12.3 Rendezvous service

The essential rendezvous service interface is small as it only has to provide methods to propagate messages[23]. The main part of the entire server connection handling is done internally and is not really of interest for a user.

Message propagation: When propagating a message, one has to provide the endpoint message, the destination service name/parameter and a TTL.

The TTL is an integer value indicating the maximal number of hops the message can be forwarded. Usually, we just set the maximum `Ttl_max` (50 hops).

There are several propagation methods: `propagate_in_group` sends the message to all connected rendezvous servers whereas `propagate_to_neighbours` uses the endpoint propagation method (propagating to the local network). `propagate` is usually the preferred method as it calls both methods above. We provide the possibility to propagate messages to a given list of peers with `propagate_to_peers`.

The rendezvous service is responsible for sending messages, the recipient endpoint service will pass the message directly to the specified service, the rendezvous service is not involved on the recipient side.

When we repropagate a message (meaning to propagate a received propagated message), the rendezvous service is able to detect this and automatically reuses the meta data stored in a special rendezvous message element.

Rendezvous events: As many services rely on message propagation, they would like to make sure, that the current peer is connected to a rendezvous so that message propagation is guaranteed. However, just at the time when the platform has started, the rendezvous connection is not available yet. So, it is rather useless at this time to use the service.

We therefore need a way to know when the connection will be of use to the services. That is what *rendezvous events* are designed for. Interested parties may register for such events. Current supported rendezvous types are only the connection and disconnection events to a rendezvous server as we currently just implement an edge peer.

To receive these events, we register an agent with `extend_rendezvous_event_handler`. The agent should expect a `P2P_RENDEZVOUS_EVENT` which provides the event type and the involved peer ID (e.g. the rendezvous server). An agent should not do time consuming processes as it would stall incoming messages from the rendezvous.

[23] See Section 4.4 for a detailed rendezvous service description.

Listing 4.15 shows how to publish the peer advertisement as soon as we are connected to the group.

Listing 4.15: Example rendezvous event handler

```
process_rendezvous_event
  (an_event: P2P_RENDEZVOUS_EVENT) is
    -- Publish our peer advertisement
    -- to group when connected to rdv
  require
    Event_valid: an_event /= Void
  do
    if
      an_event.type =
      {P2P_RENDEZVOUS_EVENT}.
      type_connected_to_rendezvous
    then
      peer_group.
        discovery_service.
        publish_advertisement_remotely
        (peer_group.peer_advertisement, Void)
    end
  end
```

Rendezvous Seeds: We have to specify the rendezvous server address (or multiple addresses) so that a new, isolated peer can contact the group. For a peer application, one would elect some peers as permanent rendezvous servers and make their addresses available.

As described in Section 4.4, there are several ways to do this. The preferred solution is to maintain a file, accessible through HTTP, with a rendezvous server list. The peer configuration then would be hard-coded to this URL. The NPG rendezvous list can be found at the following address, Listing 4.16 shows its content:
http://rdv.jxtahosts.net/cgi-bin/rendezvous.cgi?2

Listing 4.16: Public NPG rendezvous seeds

```
http://209.128.126.120:9700
http://209.128.126.120:9710
tcp://192.18.37.36:9701
tcp://192.18.37.37:9701
tcp://192.18.37.38:9701
tcp://209.128.126.120:9701
tcp://209.128.126.120:9711
```

Edge peers then randomly try one of the seed addresses and continue trying until they get a connection lease.

The rendezvous configuration contained in the platform configuration file is simple and looks like the one in Listing 4.17. VamPeer updates the configuration with known servers, once it has resolved a seed URL.

Listing 4.17: Example rendezvous configuration

```
<Parm type="jxta:RdvConfig" config="client">
  <seeds>
    <addr seeding="true">
      http://\origo.ethz.ch/rdv.cgi
    </addr>
    <addr>tcp://129.132.105.170:9700</addr>
  </seeds>
</Parm>
```

4.12.4 Resolver service

Resolver handlers: Clients using the resolver service choose a unique handler name and register a handler agent for processing query messages and one for response messages. Queries and responses are tightly coupled because when one sends a query, one is also interested in replies and a typical peer will perform queries and replies. The handler registration works with the `extend_handler` command. The handler name is usually the client service module ID.

A resolver query handler gets a `P2P_RESOLVER_QUERY` object containing the query string with some meta information. The agent should set the `repropagate` flag in the query object to specify if the resolver should repropagate the message or not. Repropagation is only done by the resolver if the peer is configured as a rendezvous. The response handlers have the equivalent signature, they expect a `P2P_RESOLVER_RESPONSE` instance. Besides queries and responses, the resolver handles also SRDI messages.

Querying and responding: To send a query, we build a `P2P_RESOLVER_QUERY` as shown in Listing 4.18[24]. Besides the query string, it needs the source peer ID, the handler name and an integer ID. The ID helps to identify responses. The resolver automatically adds the source peer route advertisement so that every recipient is able to respond directly.

[24]The final query would then look like in Listing 4.2 on page 52.

Listing 4.18: Sending a resolver query

```
send_resolver_query (a_dest: P2P_PEER_ID) is
    -- Send a resolver query
  require
    Dest_valid: a_dest /= Void and a_dest.is_valid
  local
    query: P2P_RESOLVER_QUERY
  do
    create query.
      make (peer_group.peer_id, handler_name,
      1, "Got a beer?")
    peer_group.
      resolver_service.send_query (a_dest, query)
  end
```

The example sends the query to a specific peer but it is possible to send messages to the entire group. We have to use the `propagate_query` method which only expects the query.

Responding is equivalent. It is somewhat easier to respond to a received query because `P2P_RESOLVER_RESPONSE` provides a constructor to create a response out of a query: `make_from_query`.

Note that a response does not need to be preceded by a query. This means that a response can also be propagated in order to publish information to everyone.

4.12.5 Discovery service

As we already know is the discovery service used to deal with advertisements. It provides methods for querying and publishing, locally and remotely. Most functions differ between peer, group and other (general) types of advertisements because also the discovery protocol does.

Querying: A query always consists of the advertisement type and possibly a key/ value pair. The value may be unspecified in local queries meaning to find all advertisements that have an element named like the key. Wild cards are allowed in values as already described in Section 4.4.

For local queries, we call one of the `local_*_advertisements` methods (there is one for each type) passing the key and a value. The return value is a list of all matching advertisements.

It is possible to get an advertisement from the store using its unique ID. This is done with a `local_*_advertisement` command. While passing an ID

for the peer or group advertisements, one has to specify the exact unique ID as `STRING` when looking for an other advertisement type.

Local queries can always be performed except when the discovery module is stopped or has failed during the start.

Remote queries are done with the `query_remote_advertisements` method. For this, we need to build a `P2P_DISCOVERY_QUERY` object and possibly specify a single recipient peer and a response handler. When the recipient peer is not specified, the query is propagated to the group.

A query may contain an advertisement type, a key/value pair, a threshold and a source peer advertisement. We have to pay attention to the different semantics for some attribute combinations.

Normally, we specify all attributes which means we search all matching advertisements of the given type. The threshold defines how many results we receive at most. However, we could receive a lot more since we may receive answers from various peers. A peer query example is provided in Listing 4.19.

Specifying only the type `Peer` and the threshold 0, all recipient peers should send a response with their own peer advertisement.

Listing 4.19: Sending a remote discovery query

```
discover_buddy is
    -- Send discovery query for
    -- peers named "Buddy"
  require
    Network_connected:
      peer_group.rendezvous_service.is_connected
  local
    query: P2P_DISCOVERY_QUERY
  do
    create query.make
      (peer_group.discovery_service.type_peer)
    query.set_threshold (10)
    query.set_restriction ("Name", "Buddy")
    peer_group.discovery_service.
      query_remote_advertisements
      (query, Void, agent response_handler)
  end

response_handler
  (a_response: P2P_DISCOVERY_RESPONSE) is
    -- Prints buddy IDs of all results
  require
    Response_valid:
```

```
    a_response /= Void and a_response.is_valid
local
  advs: DS_LIST_CURSOR
  [P2P_PEER_ADVERTISEMENT]
do
  from
     advs := a_response.
       all_peer_advertisements.new_cursor
    advs.start
  until
    advs.after
  loop
    print (advs.item.peer_id.out + "÷ N")
    advs.forth
  end
end
```

When the key/value pair is not specified, recipient peers should return a random advertisement set matching the given type. The set count should not exceed the threshold value.

Response handlers: In the discovery service, there are two possibilities to add a response handler. If we are interested in our query, we specify a *query ID* handler when querying. This handler will be called when a response for our query is received. Since multiple responses can be sent, the handler stays registered until we call `remove_queryid_listener`.

The other response handler type serves general responses which can be registered through `extend_response_listener`. Whenever a response is received, the discovery calls all response agents (actually, after a possible, specific query ID handler).

When no response handler is called, the discovery will itself locally publish all received results. So, whenever we register a handler, we have to handle the results and publish them, if needed.

It is only possible to register handlers for responses. Queries are always handled by the discovery service itself.

Publishing: Publishing an advertisement locally, using `publish_advertisement_locally`, means to save it in the local advertisements store under its unique ID. An older advertisement with the same ID is replaced.

On remote publishing, we may either choose to build the discovery response ourselves or to just pass a single advertisement to

`publish_advertisement_remotely`. We may specify a recipient peer, when we do not want the response to be published to the group.

A `P2P_DISCOVERY_RESPONSE` contains a list of results and the type and the key/ value pair. A responder may also provide its peer advertisement.

An advertisement is always published together with its remote expiration time. When the lifetime is not set, the default expiration time for remote peers defaults to two hours.

Note that remotely published advertisements cannot be revoked. They may be passed among group peers until they expire. Though, it is possible to delete an advertisement locally using one of the flush methods which expect the unique ID advertisement.

See Listing 4.20 to see a publishing example and how to set the lifetime.

Listing 4.20: Publishing an advertisement remotely

```
publish_advertisement
  (an_adv: P2P_ADVERTISEMENT) is
    -- Publish an advertisement
    -- setting its lifetime to a day from now
  require
    Advertisement_valid:
      an_adv /= Void and an_adv.is_valid
  do
    an_adv.set_lifetime_relative (86400000)
    peer_group.
      discovery_service.
      publish_advertisement_remotely
      (an_adv, Void)
  end
```

4.13 Writing a P2P application

While the VamPeer library offers basic P2P features, the overlying applications will have to specialize them and to design their own application messaging protocol above JXTA.

For its design, we recommend to rely on the JXTA services, especially on the discovery. The use of advertisements is a core idea in JXTA and it is recommended to create advertisements for user related entities too.

Writing a user service: Applications will usually create at least one service which closes the gap between the application logic and JXTA.

To create a module, we just inherit from the deferred class `P2P_MODULE` and effect the methods `start`, `suspend` and `stop`[25]. We usually also redefine the `init` method to create needed data structures.

We have to implement the feature `check_dependencies` which identifies the required module dependencies used for loading. It is therefore called in the precondition of `init`. We are allowed to use other services, but we have to check their availability first.

While it is simple to build a user service, the integration into the peer group involves several steps.

As the user service is a module, we have to build a MCID, a MSID and a module implementation advertisement. Listing 4.9 already pointed out how to do this.

Because we change the set of user services in the group, we will certainly have to build a private peer group as described in Section 4.11.1. We now also have to specialize the peer group implementation by creating a new peer group class which inherits from `P2P_GENERIC_PEERGROUP`. `define_modules` and `load_extern_module` should be redefined as shown in Listing 4.21.

Listing 4.21: Redefining peer group modules

```
define_modules is
    -- Define Group services in 'modules_list'
  do
    -- add standard modules from
    -- parent peer group to 'modules_list'
    Precursor
    -- add user service to the end of 'modules_list'
    modules_list.put_last
       (["user_service", service_mcid,
       service_msid,
       parent_is_owner_if_available])
  end

load_extern_module
  (an_id: P2P_ID;
  a_mia: P2P_MODULE_IMPLEMENTATION_ADVERTISEMENT;
  a_name: STRING): P2P_MODULE is
    -- Load user module
  do
    if
      service_msid.is_equal (a_mia.specification_id)
    then
```

[25] Please read Section 4.6.1 for further details.

```
      create {SERVICE_CLASS}
        Result.init (Current, an_id, a_mia)
    else
      Result := Precursor (an_id, a_mia, a_name)
    end
  end
```

The `parent_is_owner_if_available` flag means that the module loader will only create a new module loader if the parent peer group did not define this module. If each peer group would need its own instance, we would specify `current_is_owner`.

Using the new private peer group implementation, our user service will be loaded and managed by VamPeer.

4.14 Examples

The current release of VamPeer also contains some examples that new users can use to get into the code and to test the library. We describe them here and show how to run them.

Endpoint message sender/handler: The first example shows how to use the *endpoint service*. It was originally created when the other services were not available yet. Therefore, it does not make use of the discovery to find other peers.

It consists of two parts, namely two peers: the `endpoint_message_sender` and the `endpoint_message_handler`, which both run in the public NPG. The sender will send a simple endpoint message to the handler peer which sends a reply message back. Both peers log the events to the standard output, so we can see what is currently happening. While the handler only quits when we shut it down (with `Ctrl-C`), the sender terminates as soon as it receives the reply.

The handler's platform is configured to listen on port `9710` while the sender chooses its port automatically. To start the sender, we have to specify the handler peer IP and port. This is not the way we would do it with other peers in our application. We want like to limit our example to the endpoint service possibilities.

To run the example locally, we start the endpoint message handler and afterwards the sender application with the arguments: `localhost9710`. The logging level is set to `INFO`, so we can see when the platform has been started and the message has been sent and received.

Looking at the code, we see how the sender creates and sends an endpoint message and how it registers the service listener to receive the reply. The handler looks similar but also uses the endpoint filter method to display all incoming messages.

The example is very simple but shows how the minimal configuration for a VamPeer application.

4.14.1 Rendezvous propagation

The next example demonstrates the rendezvous propagation mechanism. It resides in the `rendezvous_propagate` example directory. The application connects to a NPG rendezvous and propagates a message every five seconds.

When running multiple instances, we can see the propagated messages from other peers. But it does not work; messages are not propagated to the entire group. The reason for this is, that we would have to adapt the rendezvous server to repropagate messages for our specific service[26].

Thus, this example shows how we can not use the rendezvous service for propagating any message and it shows how important it is that the rendezvous server is part of the P2P system and not just a standard JXTA infrastructure peer.

4.14.2 Discovery

The `discovery` example shows how we can find other peers in the public NPG. We just run multiple instances and they should be able to discover each other. The application only expects a configuration directory path as argument.

The details of the discovery procedure are as follows: As soon as we are connected to a rendezvous, we publish our peer advertisement in the group. We then send one peer query request and wait one minute for responses. Incoming peer advertisements are printed out immediately.

When no other peers are known to the rendezvous, we should at least receive the rendezvous peer and our own advertisement. Note that we may get responses from multiple peers, not only from the rendezvous because the rendezvous propagates the query request also to the others.

[26] See Section 4.6.8 for more details to this.

4.14.3 JXTA JSE rendezvous server

The last two examples make use of the NPG rendezvous servers. But we mentioned already that the public infrastructure is not accessible at the moment. Hence, we will have to run our own NPG rendezvous server. The `RdvServer` example shows how we can set it up using the JXTA's reference implementation JSE in version 2.4.1.

The Java application provides a rendezvous and a relay server and is configured to listen on port 9700 on all interfaces. When we start the script `startNPGPeer.sh`, the rendezvous will be started in the NPG.

To point the discovery example to our new rendezvous server, we have to change its platform configuration. We normally start the discovery application first to generate the configuration directory. Then, we are able to change the configuration XML file by replacing the old seeds and the NPG seed URL with the new destination address.

Chapter 5

Search mechanisms

Various systems and models have been described in the past for coordinating components in distributed settings. This section starts by presenting a simple abstract model of lookup, and then relating that model to a set of predating approaches. This work has been published as a paper [4] and is used for search in Origo Core.

5.1 Lookup model

Components are described towards the outside world by respective *specifications* (see Figure 5.1). Lookup services basically provide components, they are a means to construct and advertise such specifications and they are also a mechanism to query components based on (specification) *templates*. The composition and nature of these specifications and templates, as well as the *matching* between them, vary between approaches.

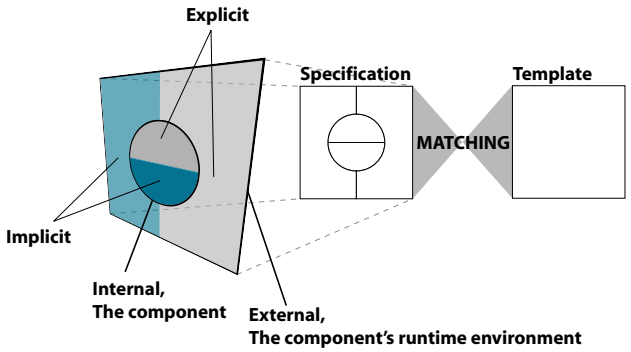


Figure 5.1: Component and lookup model

Internal vs. external specification: When viewing specifications as being based on different properties, one can in a first step distinguish between *internal* and *external* properties. Internal properties are based on the nature of components themselves, i.e., they reflect properties of a given component. External criteria reflect properties which pertain to the surroundings of the component, such as its context or (runtime) environment.

Implicit vs. explicit specification: In a second step, one can distinguish between *implicit* and *explicit* properties. The former kind of criteria reflect intrinsic properties *of the services provided* by a given component; they are not influenced by the nature and set of targeted clients for that component, or the means by which the component is made available to such consumers. Explicit criteria in contrast, manifest in the way the component's very design is influenced by the perspective of making it ultimately available to the outside world.

Static vs. dynamic evaluation: Furthermore, the evaluation of the matching can be *static*, i.e., based on attributes of component specifications which are evaluated once and for all when the component is loaded, or *dynamic*, in which case the matching becomes a continuous process (see Section 5.6).

5.2 Examples

We illustrate the above model through a set of well-known lookup services, and overviewing derivatives for each. Results are summarized in Table 5.1 (due to the sparse occurrence of dynamic criteria in common lookup services the distinction static/dynamic is however omitted).

Domain Name System (DNS): DNS is very likely to be the most frequently used, static, name-based lookup system. Components are IP addresses, the specifications are (internal) host names, the templates are host names as well, and the matching tries to find the component that registers with a given host name (explicitly) and returns it's IP if possible.

Network Information Service (NIS): NIS is one of the oldest type-based, static lookup systems. Components are the entries of the maps (external), the specifications are map names (implicit), the templates are either map names or nicknames (e.g., *passwd* for *passwd.byname*), and the matching is the result of the `ypcat` command.

CORBA: The Common Object Request Broker Architecture (CORBA) [16] defines both a Naming Service and Trading Object Service for name-based and type-based lookup of objects respectively. The Naming Service represents the original means of looking up objects based on a *hierarchical* naming scheme, where an object is registered (explicit) and made available by attaching it (external) a unique name $N_1.\cdots.N_n$ of which each component N_i is a name/kind-pair. In this case, specifications and templates are both defined as sets of such pairs. Names for the Java RMI registry, or regular expressions, are similar in that sense, with $n = 1$.

The Trading Object Service offers rich combinations of means of defining the service type of a component. The most preferred way of attaching a type specification to a component consists in attaching it a name/value-pair. This definition of a component is external and explicit as well: the "type" describes actual properties of the component itself, but is not implicit like the actual classification of a component according to the type system of the considered language/environment.

Note that the OMG has more recently specified the Interoperable Name Service, defining URL-format object references that can be typed into a program to reach services at a remote location, including the Naming Service.

RM-ODP: The Reference Model for Open Distributed Processing (RM-ODP) [77] defines, similarly to CORBA, both a "white pages" (name-based) and "yellow pages" (type-based) lookup service (both explicit and external), going by the names of relocator and trader respectively. The latter service describes two roles which interacting components may take: *exporters* of services, and *importers*. A *service description* is an interface (type) and a set of properties attached to it, and a *service offer* binds a service description to a concrete component, which can be a CORBA object or another object. Properties are thus used to describe specifications and templates, the latter ones being more precisely combinations of properties; rules are expressed based on properties and operators (these are called matching criteria).

A novelty of the trader specification is the description of delegation and collaboration among individual trader units, which however does not seem to impact the model ultimately perceived by an application programmer, as, expressed in our terminology, specifications are simply cascaded.

UDDI: The *universal description, discovery and integration* (UDDI) [78] specification defines a lookup service for web services. Such a *registry* is centered around a *public cloud*, a set of replica nodes storing white pages (abstract services by "name"), yellow pages (by "type"), and green pages

(by "description" and "location"). Targeting at web services, UDDI encompasses a set of XML messages for SOAP-based interaction with registries. Each party is described through a *business entity*, several of which can be linked through *publisher assertion*s. A *business service* is a particular web service offered by a business entity. Such a service is described by one or more *binding templates*, which optionally contain textual service descriptions, and URLs for the respective services. Finally, *binding templates* refer to one or more *tModel*s, which contain the pointers to actual descriptions of the services offered, and delineate the interaction protocols with the respective services. All the above-mentioned entities describe a refined pattern for specifications in the sense of our model introduced before-hand. The enforcing of authentication is covered in our model by external explicit criteria (see Section 5.5.5). The load distribution among nodes forming the public cloud is achieved in our implementation in an efficient manner by distributing the matching, greatly transparently, over a peer-to-peer overlay network (see Section 5.6).

Note that UDDI is a rare example of dynamic lookup, where components can be notified of changes in specifications of other components. Further examples are given by load balancing, or reuse frequency [79].

Service Groups: Sadou et al. [14] introduce a notion of *service group* to mediate between client and server components. These are motivated by the desire for type evolution, e.g., the possibility of adding parameters to methods. Just like in RM-OPD, the approach introduces both a notion of *type* which reflects provided services (i.e., the server side) in the terminology introduced by the authors, and a notion of *role* which represents the needs of customers (i.e., the client side).

At a first glance, one could hence be brought to viewing the types of [14] as specifications in our case, roles as templates, and service groups as defining the matching, respectively. However, the emphasis of [14] consists in making services of a given type available to clients expecting a slightly different type. Service groups are thus a form of glue aiming at expressing *how to pass from a given type to a given role*. They consist in stubs for respective server objects, which transform invocations based on a given role (the expected type) such as to fit the effective type. In our model, this represents explicit, internal component registration, and the specifications are made up of the stubs.

In a sense, HydroJ [11] and LuckyJ [12], can be seen as similar approaches to service groups, as these are also based on some notion of type. Borrow/Lend [80], a derivative of the Type-based Publish/Subscribe (TPS) abstraction [81], as suggested by the name of the latter paradigm, in contrast,

Criteria	Explicit	Implicit
External	UDDI, CORBA Naming, Trading, Java RMI, Linda, Regular Expressions, Tagged Sets, Borrow/Lend, SecOS	Reuse Frequency, Load Balancing, NIS
Internal	HydroJ, LuckyJ, Service Groups, DNS	Method Dispatch, Borrow/Lend

Table 5.1: Coarse classification of lookup services

is primarily based on type-based matching of inherent Java object types (implicit, internal). The types are augmented by (dynamic) predicate evaluation, and with keys (explicit, external).

Coordination Spaces: The Borrow/Lend abstraction can in fact be seen as a variant of the Linda Tuple Space [9] with callback functionalities. The original Tuple Space is a means of exchanging information among distributed components, based on tuples of place holders (types) and values, i.e., a mixture of value-based and type-based matching, where values can also be character strings. This demonstrates how thin the border between types and values is.

Just like Borrow/Lend, Tagged Sets [82] are a variant of Tuple Spaces, where tuple items can also be predicates (leading to a dynamic evaluation), or keys (symmetric or asymmetric). Similarly, SecOS [83], supports the use of keys, with a partial matching. Clearly, any such criterion is explicit and external.

5.3 A note on values and types

A distinction that is often made when discussing component lookup is the one between *values* and *types*. This is nicely illustrated by the metaphors of "white pages" and "yellow pages" respectively.

However, component lookup in a distributed heterogeneous environment is basically untyped. Matching components for their "type" boils down to matching such components for the *name of their type*, an internal property of these components. The possibility of registering several objects under a same given name, as supported by many systems, illustrates this seamless transition; by doing so, such a name becomes more a type description than a unique identifier. The issue of matching in such a setting becomes essentially a question of *depth*, in a way similar to the issue of object copy-

ing/cloning [84]. Any categorical distinction between values and types at this level seems unnatural. This is captured by our abstract notions of specifications and templates, which will become clearer through the matching model presented in Section 5.4, and illustrations thereof in Section 5.5.

5.4 Matching model

The matching model presented in this section has resulted from the desire of capturing all the different lookup criteria outlined in the previous section.

In our model, the matching of components against requirements builds on the two basic notions introduced in the previous section, namely *specifications* and *templates*. The former roughly represent actual component descriptions (i.e., server-side views of components, see Figure 5.1), and the latter represent requirement descriptions (i.e., client-side views of components). In our matching model, specifications and templates are related by *matching modules*. Our goal is to be able to combine several specifications and templates into a compact notation and to design a lookup mechanism that sorts the retrieved components in a list.

Our solution relies on mathematical formulae containing templates. As an example the formula $t_0 + 3.0 - t_1 * t_2$ combines the three templates t_0, t_1 and t_2. Such a formula will be evaluated for each component C that has specifications s_0, s_1 and s_2 respectively corresponding to each template. The evaluation replaces each template with a value (the matching value) that is calculated by applying a matching function ($?_i$) between the specifications of the component and the templates. As an example evaluating the formula with given specifications will return the evaluation of:

$$(?_0(s_0, t_0) + 3.0 - ?_1(s_1, t_1) * ?_2(s_2, t_2))$$

For each component, this formula yields its matching value. When a client looks a component up, it is given a list of components sorted by their matching values in descending order. Components for which the matching value is 0 or below are omitted from the list. In the remainder of the section we define the theoretical framework to formalize this intuition using denotational semantics.

5.4.1 Matching modules

A *matching module* is a triplet encompassing a set of specifications $\mathfrak{S}$, a set of templates $\mathfrak{T}$ and a matching relation ?.

$$\begin{array}{rcl} mm & ::= & (\mathfrak{S}, \mathfrak{T}, ?) \\ & & where\ ? : \mathfrak{S} \times \mathfrak{T} \rightarrow \mathbb{N} \end{array}$$

5.4.2 Specifications

A specification $\mathfrak{S}$ is itself a set of *specification terms* s_i. Informally, a specification term is the specification for a component according to a given formalism. A template $\mathfrak{T}$ is itself a set of *templates terms* t_i. Informally, a template term delineates a set of components according to a given formalism. The matching relation ? is a function that takes a specification term and a template term as arguments and returns a natural number.

In short, we define here what we need for providing ways of matching specifications and templates. Our goal being to integrate several of these modules into a multi-module specification, we do not enter into details but rather give examples of this in Section 5.5.

5.4.3 Qualified specifications

A *qualified specification term* qs is a specification term annotated with a *qualifier*.

$$\begin{array}{rcl} s & \in & \mathfrak{S}_i \\ val & ::= & n \in \mathbb{N} \\ comp & ::= & < \mid > \mid \neq \mid \leq \mid \geq \mid = \\ qualifier & ::= & \textbf{required}\ comp\ val \mid \varnothing \\ qs & ::= & s\ qualifier \end{array}$$

Qualifiers on specification terms are used as a way for the component provider to order differences in the treatment of the matching. We specify two different types of qualifiers: $\varnothing$ that means that we do not modify the basic mechanism (that we always omit in practice as a notation abuse) and **required** that allows us to filter and impose a condition on the matching for specific specification terms. This latter qualifier allows us, in particular to envision security-constrained matching as shown in Section 5.5.5. Even if, for now, we only consider the qualifiers **required** and $\varnothing$ we could imagine other qualifiers that modify the infrastructure's behavior accordingly.

A *component specification* CS consists of a set of qualified specification terms that appear at most once in the set of specifications of a given matching

module.

$$\begin{array}{lll} CS & ::= & \{qs_1,\ ...,\ qs_n\} \\ & & such\ that\ \forall\ i,\ j\ \in\ [1,n]\ \ s_i \in s_0\ s_j \in s_0\ \Rightarrow\ i = j \end{array}$$

A component specification is the way a component provider can describe its components.

5.4.4 Templates

A *template* $T \in \mathfrak{T}$ consists of a mathematical formula using mathematical operators and *template terms*.

$$\begin{array}{rll} t & \in \mathfrak{T}_i & \\ op & ::= & +\ |\ -\ |\ *\ |\ / \\ T & ::= & n \in \mathbb{N}\ |\ t\ |\ T\ op\ T \end{array}$$

The idea is, that unlike qualified specification terms that are composed in a list to make the component specification, we compose template terms to a mathematical formula in order to allow component seekers to allocate more weight to some specification. It also allows to exclude components that answer to a specification by using subtractions and divisions to lower their matching values and possibly rule them out of the returned list.

5.4.5 Matching

The *valued matching* of a component specification CS with a template T consists in matching on the specification and calculating its value according to the template definition. It is defined as follows:

$$valuedMatch \quad ::= \quad CS?_v T$$

$$\begin{array}{lcl} \mathcal{V}[\![\cdot]\!] : valuedMatch \rightarrow \mathbb{Q} \cup \{\infty\} & & \\ \mathcal{V}[\![CS?_v n]\!] & = & n \\ \mathcal{V}[\![CS?_v t]\!] & = & 0\ if\ \nexists\ qs = s_0\ q_0 \in CS \\ & & \quad such\ as\ \exists\ mm_0 = (\mathfrak{S}_0, \mathfrak{T}_0, ?_0)\ |\ t \in \mathfrak{T}_0,\ s_0 \in \mathfrak{S}_0 \\ & & ?(s,\ t)\ otherwise \\ \mathcal{V}[\![CS?_v T_1\ op\ T_2]\!] & = & \mathcal{V}[\![CS?_v T_1]\!]\ op\ \mathcal{V}[\![CS?_v T_2]\!] \end{array}$$

The intuition behind the matching we describe is the following: each template term within the mathematical formula of the template is replaced by the result of the application of the matching relation between the template term and the specification term of the component specification.

The *matching compliance* of a component specification CS with a template T describes the specification terms matched. It is defined as follows:

$$
\begin{array}{lcl}
compliesToMatch & ::= & CS?_cT \\
\\
\mathcal{C}[\![\cdot]\!] : compliesToMatch \rightarrow \mathbb{B} & & \\
\mathcal{C}[\![\{s \textbf{ required } comp_0\ n_0\}?_cT]\!] & = & \text{TRUE } if\ \exists\ t\ in\ T\ s.a.\ \mathcal{V}[\![s?_vt]\!]\ comp_0\ n_0 \\
 & & \text{FALSE } otherwise \\
\mathcal{C}[\![\{s\varnothing\}?_cT]\!] & = & \text{TRUE} \\
\mathcal{C}[\![\{qs_1,\ ...,\ qs_n\}?_cT]\!] & = & \mathcal{C}[\![\{qs_1\}?_cT]\!] \wedge ... \wedge \mathcal{C}[\![\{qs_n\}?_cT]\!]
\end{array}
$$

As a simple explanation, a template complies with a specification if all the required conditions on the specifications are fulfilled by any of the basic templates.

5.4.6 Component selection

Finally, we can define the *selection* mechanism built on top of the valued matching and the matching compliance. A component C declares its interface in its component specification CS. The component repository $\mathfrak{C}$ consists in a set of components stored with their specifications. These can be selected using the selection operator $\downarrow$ that returns a list of components for which we show the semantics $\mathcal{E}$.

$$
\begin{array}{lcl}
\mathfrak{C} & ::= & \{(CS_1, C_1), ..., (CS_n, C_n)\} \\
lookup & ::= & \mathfrak{C} \downarrow T \\
\\
\mathcal{E}[\![\cdot]\!] : lookup \rightarrow list\ of\ (CS_i, C_i) & & \\
\mathcal{E}[\![\mathfrak{C} \downarrow T]\!] & = & \{(CS'_1, C'_1), ..., (CS'_m, C'_m)\} \subseteq \mathfrak{C} \\
 & & such\ that \\
 & & \forall i \in [1,m], \mathcal{C}[\![CS'_i?_cT]\!]\ and\ \mathcal{V}[\![CS'_i?_vT]\!] > 0 \\
 & & and\ \forall i,j \in [1,m],\ i < j \Leftrightarrow \mathcal{V}[\![CS'_i?_vT]\!] \geq \mathcal{V}[\![CS'_j?_vT]\!]
\end{array}
$$

Intuitively, the final result of a component selection on a repository is a list containing elements from the repository ordered by decreasing matching values. That way, we can obtain the component that is best adapted regarding to the templates we defined. In the next section we show examples of such matching modules and how they can be used.

5.5 Illustration

This section illustrates our generic model of component lookup through a small set of existing lookup schemes. More examples can be found in a longer

version of this paper [85] (e.g. examples based on nominal and structural subtyping or on reuse frequency [79]).

5.5.1 Unique identifiers

As a first simple example, we consider the selection mechanism based on a unique component identifier. In that case the matching module can be described by the following triplet:

$$\begin{array}{lll} mm_{\mathfrak{UID}} & ::= & (\mathbb{N}, \mathbb{N}, ?_{\mathfrak{UID}}) \\ & & where\ ?_{\mathfrak{UID}}: \quad \mathbb{N} \times \mathbb{N} \mapsto \{\ 0,\ 1\ \} \\ & & ?_{\mathfrak{UID}}(x,y) \ = \quad 1\ if x = y \\ & & \qquad\qquad\quad 0\ otherwise \end{array}$$

As a first example of use, we can imagine a collection of software components that have unique identifiers:

$$\mathfrak{C} = \{(\{1_{\mathfrak{UID}}\}, C_1), ..., (\{1337_{\mathfrak{UID}}\}, C_{1337}), ..., (\{n_{\mathfrak{UID}}\}, C_n)\}$$

Looking up component identified by number 1337 can be made as follows:

$$\mathfrak{C} \downarrow 1337_{\mathfrak{UID}} = \{(\{1337_{\mathfrak{UID}}\}, C_{1337})\}$$

Note that a variation of this module can be used to describe the DNS.

5.5.2 Regular expressions

Among the most widespread and popular descriptions of components are component APIs, and component documentation. One can imagine selecting components based on criteria expressed on their textual description, in addition to other specifications. An example is selecting components according to their author(s), as appearing in the documentation. This constitutes the case of matching regular expressions (note that we use the original regular expressions as defined in Kleene algebra):

$$\begin{array}{lll} char & ::= & a \mid ... \\ string & ::= & char \mid string\ string \\ expr & ::= & \varnothing \mid char \mid (expr\ expr) \mid (expr + expr) \mid expr* \\ mm_{\mathfrak{regexp}} & ::= & (string, expr, ?_{\mathfrak{regexp}}) \\ & & where\ ?_{\mathfrak{regexp}} : string \times expr \mapsto \mathbb{N} \\ & & ?_{\mathfrak{regexp}}(s,e) = number\ of\ occurrences\ of\ s\ in\ e \end{array}$$

Now imagine that a user wants to obtain a component for which John Doe is indicated as the main author of that component in the accompanying

documentation and preferably take the component with the unique identifier 1337. A collection including such a component could then be:

$$\begin{aligned}\mathfrak{C} = \quad \{ \quad & (\{1_{\mathfrak{UID}}, \text{“}...author : John\ Doe...\text{”}_{\mathfrak{regexp}}\}, C_1), ... \\ & (\{1337_{\mathfrak{UID}}, \text{“}...author : John\ Doe...\text{”}_{\mathfrak{regexp}}\}, C_{1337}), ... \\ & (\{n_{\mathfrak{UID}}\}, C_n)\}\end{aligned}$$

Looking up a component fulfilling at least one of these characteristics would then produce:

$$\begin{aligned}&\mathfrak{C} \downarrow (1337_{\mathfrak{UID}} + \text{“} * author\ :\ John\ Doe * \text{”}_{\mathfrak{regexp}}) = \\ &\{ \quad (\{1337_{\mathfrak{UID}}, \text{“}...author : John\ Doe...\text{”}_{\mathfrak{regexp}}\}, C_{1337}), \\ &\quad (\{1_{\mathfrak{UID}}, \text{“}...author : John\ Doe...\text{”}_{\mathfrak{regexp}}\}, C_1)\}\end{aligned}$$

Looking up a component fullfilling both criteria can be made as follows:

$$\begin{aligned}&\mathfrak{C} \downarrow (1337_{\mathfrak{UID}} * \text{“} * author : John\ Doe * \text{”}_{\mathfrak{regexp}}) = \\ &\{ \quad (\{1337_{\mathfrak{UID}}, \text{“}...author : John\ Doe...\text{”}_{\mathfrak{regexp}}\}, C_{1337})\}\end{aligned}$$

5.5.3 Load balancing

Another criterion of component linking, is its current load.

$$\begin{aligned}mm_{\mathfrak{load}} \quad ::= \quad & (\mathbb{N},\ \varnothing,\ ?_{\mathfrak{load}}) \\ & where\ ?_{\mathfrak{load}} : \mathbb{N} \times \varnothing \mapsto \mathbb{N}^+ \\ & ?_{\mathfrak{load}}(n) = number\ of\ components\ currently\ using\ component\ n\end{aligned}$$

Imagine that a user wants to obtain the component which is currently experiencing the smallest load written by John Doe. Suppose also that some components support only up to 10 clients at the time. A collection containing such components could then be specified as follows:

$$\begin{aligned}\mathfrak{C} = \{&(\{1_{\mathfrak{UID}}, \text{“}...author : John\ Doe...\text{”}_{\mathfrak{regexp}}, C_1), ... \\ &(\{1337_{\mathfrak{UID}}, \text{“}...author : John\ Doe...\text{”}_{\mathfrak{regexp}}, \textbf{required}\ 1337_{\mathfrak{load}} < 10.0\}, C_{1337}), ... \\ &(\{n_{\mathfrak{UID}}, n_{\mathfrak{load}}\}, C_n)\}\end{aligned}$$

A programmer wishing to get such a component should perform the following lookup (note that the result is dependant of the number of clients currently connected to both components):

$$\begin{aligned}&\mathfrak{C} \downarrow (\text{“} * author : John\ Doe * \text{”}_{\mathfrak{regexp}}/(1.0+_{\mathfrak{load}})) = \\ &\{ \quad (\{1_{\mathfrak{UID}}, \text{“}...author : John\ Doe...\text{”}_{\mathfrak{regexp}}, C_1), \\ &\quad (\{1337_{\mathfrak{UID}}, \text{“}...author : John\ Doe...\text{”}_{\mathfrak{regexp}}\}, C_{1337})\}\end{aligned}$$

5.5.4 Compliance to an interface

It very often happens that programmers want to obtain components that comply to a given interface. Informally, compliance to an interface is expressed in terms of a structural subtyping relationship. Suppose that I_1 is compliant to I_2 if and only if I_1 has at least the same procedures as I_2.

$$
\begin{array}{lll}
p & & \textit{procedure names} \\
t & & \textit{types names} \\
\textit{procedure} & ::= & (p, \{t_0, ..., t_n\}) \\
I & ::= & \{\textit{procedure}_1, ..., \textit{procedure}_n\} \\
mm_{\mathfrak{comply}} & ::= & (Interfaces,\ Interfaces,\ ?_{\mathfrak{comply}}) \\
 & & \textit{where } ?_{\mathfrak{comply}} : Interfaces \times Interfaces \mapsto \{0,\ 1\} \\
 & & ?_{\mathfrak{comply}}(I_1, I_2) = 1\ iff\ I_2 \subseteq I_1,\ 0\ otherwise
\end{array}
$$

Supposing that some components offer procedures to set and get their internal attributes, the collection of components could be:

$$
\begin{array}{l}
\mathfrak{C} = \{(\{1_{\mathfrak{Wood}}, \{set\_a\ \{Void,\ string\}, get\_a\ \{string\}, decrement\ \{\}\}_{\mathfrak{comply}}, C_1), ... \\
\quad (\{1337_{\mathfrak{Wood}}, \text{“}...author : John\ Doe...\text{”}_{\mathfrak{regexp}}\}, C_{1337}), ... \\
\quad (\{n_{\mathfrak{Wood}}, n_{\mathfrak{load}}, \{set\_a\ \{Void,\ string\}, get\_a\ \{string\}\}\}_{\mathfrak{comply}}, C_n)\}
\end{array}
$$

Then a program seeking for components that comply to an interface containing set_a and get_a could make the following lookup:

$$
\begin{array}{l}
\mathfrak{C} \downarrow \{set\_a\ \{Void,\ string\}, get\_a\ \{string\}\}_{\mathfrak{comply}} = \\
\{\ \ (\{1_{\mathfrak{Wood}}, \{set\_a\ \{Void,\ string\}, get\_a\ \{string\}, decrement\ \{\}\}_{\mathfrak{comply}}, C_1), \\
\quad (\{n_{\mathfrak{Wood}}, n_{\mathfrak{load}}, \{set_a\ \{Void,\ string\}, get_a\ \{string\}\}\}_{\mathfrak{comply}}, C_n)\}
\end{array}
$$

Variants of this example are countless as we could return the number of procedures in common, or the number of lacking procedures etc. However this is the simplest variant and it corresponds to the approach of service groups [14].

5.5.5 Secure linking

By specifying a **required** clause, a component *provider* can enforce the matching of a specification as a necessary precondition for handing out any reference to its component. Our current example is presenting encrypted matching and can be considered as a subset of tagged sets [82] or any other matching mechanisms driven or restricted by encryption [83, 80].

We call $E(K, value)$ the encryption and $D(K, value)$ the decryption, for which we give the semantics $\mathcal{S}[\![\cdot]\!]$ that we detail below.

$$
\begin{array}{lll}
\overline{SKey} & & SymetricKeys \\
\overline{AKey} & & Asymmetric\ Keys\ (private) \\
AKey & & Asymmetric\ Keys\ (public) \\
value & ::= & basic_value \mid value_{\overline{AKey}} \mid value_{SKey} \\
e & ::= & value \mid E(SKey, e) \mid E(AKey, e) \mid D(SKey, e) \mid D(AKey, e)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{S}[\![\cdot]\!] :\ e \mapsto value \\
\mathcal{S}[\![value]\!] = value \\
\mathcal{S}[\![value]\!] = value \\
\mathcal{S}[\![E(SKey, e)]\!] = \mathcal{S}[\![e]\!]_{SKey} \\
\mathcal{S}[\![E(\overline{AKey}, e)]\!] = \mathcal{S}[\![e]\!]_{\overline{AKey}} \\
\mathcal{S}[\![D(SKey, e_{SKey})]\!] = \mathcal{S}[\![e]\!] \\
\mathcal{S}[\![D(AKey, e_{\overline{AKey}})]\!] = e
\end{array}
$$

The associated matching module is then:

$$
\begin{array}{lll}
mm_{\mathfrak{Crypt}} & ::= & (Keys,\ Keys,\ ?_{\mathfrak{Crypt}}) \\
 & & where\ ?_{\mathfrak{Crypt}} : \quad Keys \times Keys \mapsto \{0, 1\} \\
 & & \qquad ?_{\mathfrak{Crypt}}(K_1, K_2) = \quad 1\ if\ \mathcal{S}[\![D(K_2, E(K_1, value))]\!] = value \\
 & & \qquad\qquad\qquad\qquad\quad 0\ otherwise
\end{array}
$$

A collection containing components being locked by an asymmetric key $AKey$ could then be :

$$
\begin{array}{l}
\mathfrak{C} = \{(\{1_{\mathfrak{UID}}, \text{“}...author : John\ Doe...\text{”}_{\mathfrak{regexp}}, C_1), ... \\
\quad (\{1337_{\mathfrak{UID}}, \text{“}...author : John\ Doe...\text{”}_{\mathfrak{regexp}}, \mathbf{required} AKey_{\mathfrak{Crypt}} = 1.0\}, C_{1337}), ... \\
\quad (\{n_{\mathfrak{UID}}, n_{\mathfrak{load}}, AKey_{\mathfrak{Crypt}}\}, C_n)\}
\end{array}
$$

A programmer wishing to know all the components locked with $AKey$ should then make the following lookup:

$$
\begin{array}{l}
\mathfrak{C} \downarrow \overline{AKey}_{\mathfrak{Crypt}} = \\
\quad \{(\{1337_{\mathfrak{UID}}, \text{“}...author : John\ Doe...\text{”}_{\mathfrak{regexp}}, \mathbf{required} AKey_{\mathfrak{Crypt}} = 1.0\}, C_{1337}) \\
\quad (\{n_{\mathfrak{UID}}, n_{\mathfrak{load}}, AKey_{\mathfrak{Crypt}}\}, C_n)\}
\end{array}
$$

Implementation-wise, locking a component with a cryptographic key means that the access to the component should be made on the platform where the component is located. Similarly to tagged sets [82], the keys do not need to transit through the network.

5.6 Implementation

This section first presents our Eiffel implementation of the model described in Section 5.4. Thereafter, we show how to use the implementation of $\mathcal{COLOS}$in practice.

The implementation of the $\mathcal{COLOS}$ model consists mainly in the specifications, templates and the surrounding component infrastructure. Currently the framework consists of 21 classes with 1700 lines of code altogether. We are extending it to more component models and plan on making it available as open-source.

Specifications: `LL_SPECIFICATION` is a list of `LL_SPECIFICATION_TERMs`. The deferred (abstract) class `LL_SPECIFICATION_TERM` should be subclassed by a programmer who wants to define his own matching module. The only mandatory feature to be implemented returns a `STRING` representing the name of the corresponding matching module. The infrastructure already implements the features to look through the specifications given that the `LL_SPECIFICATION_TERMs` return the correct matching module name. This enables an implementation based on hashtables. Just like for templates, which are described in following Section, we use the possibility to define our own infix operators for setting constraints on the specifications that describe a component. The Eiffel programming language makes it easy to define these operators and together with automatic conversion functions they allow writing easily readable code.

Templates: To implement our prototype, we relied on two features of the Eiffel language, namely (1) user-defined infix operators and (2) user-defined automatic type conversion. Infix operators allow us to compose templates using the infix operators as defined by the natural mathematical intuition while automatic conversion lets us have valid types for general mathematical operations. According to the latest definition of Eiffel and the priority of the operators, the usual priorities apply. The infix operators are coded into `LL_TEMPLATE` and are thus inherited by all templates. The automatic conversion from `DOUBLE` to `LL_TEMPLATE` ensures that we can compose doubles and templates in a same expression containing infix operators. In short, the Eiffel compiler (ISE Eiffel 5.7) converts mathematical formulae containing templates by transforming the doubles that they contain into `TEMPLATES`. As an example, the formula

```
template:= 2.0*template0-1.0/(template1-template2)
```

is automatically transformed by the compiler into:

```
template:=
   ((create {LL_TEMPLATE}.make_from_double(2.0))*template0)
      -
   ((create {LL_TEMPLATE}.make_from_double(1.0))/(template1
      -template2))
```

The deferred class `LL_TEMPLATE_TERM`, inherits from the class `LL_TEMPLATE`. A programmer wishing to implement a matching module should subclass it and implement the feature `match` that takes an `LL_SPECIFICATION_TERM` as an argument and he should also provide a feature returning the name of the matching module as mentioned previously. Note that in our infrastructure the only `LL_SPECIFICATION_TERMs` that can be passed as parameter to the `match` feature are the ones actually belonging to the same matching module.

Decentralized lookup. The current matching prototype infrastructure performs centralized component lookup. We are currently in the process of augmenting our implementation for efficient component lookup in peer-to-peer (P2P) settings, which will make our infrastructure available as a service within a peer group of JXTA networks [2].

In order to complete such a decentralized lookup efficiently, it is very useful to be able to "decompose" the matching. The idea can be viewed as a generalization of the problem of content-based event routing in P2P networks, where event contents are viewed as consisting in several properties which are each matched against values, and an overlay network can be built which regroups participants with common interests and whose nodes many perform matching of only subsets of the properties (e.g. [86]).

In order to be able to decompose the matching in the lookup problem, a little help is however required from the programmer. Both specifications and templates have to provide access to a tree-based representation of themselves, akin to abstract syntax trees. The individual tree nodes represent elementary matching operations, and can be performed in a decentralized, yet minimally redundant, manner.

The logical regrouping of several *tModel*s to a *bindingTemplate*, several *bindingTemplate*s to a *businessEntity*, and several instances of latter kind to a *businessService* in UDDI (see Section 5.2), is but an illustration of such a decomposition.

5.6.1 Using the library

In the current state of the implementation of $\mathcal{COLOS}$, a programmer wishing to use the component lookup mechanism can simply instantiate the class

`LL_COMPONENT_COLLECTION` and the components along with their specifications. By subclassing the two deferred (abstract) classes `LL_SPECIFICATION_TERM` and `LL_TEMPLATE_TERM`, the programmer can implement a matching module. It implies setting two variables and redefining the feature *match*. Note that keeping a reference to the object encapsulating a component with its specification allows revoking parts of the specification dynamically.

In the following example (see Figures 5.2 and 5.3) we show how one describes a component and then uses our lookup mechanism to match requirements against the entire component repository. We see how the specification terms are first declared and enriched with the corresponding information. Then they are added to the component's specification. Note how the *less* operator is used to impose a constraint on the specification about the component's load. In the second Listing (see Figure 5.3) of the example it is shown how to prepare a component lookup. Instead of specifications we are now preparing templates that are put together to match against the component repository. The ^-operator is used to initiate the matching. In the resulting list the components are ordered according to rating of the matching in respect to the template formula. In this case we are only interested in the component with the highest rating and we are therefore only obtaining the first component of the resulting list.

5.7 Conclusions

Lookup mechanisms are an essential part of the very foundations of distributed component interaction. Various systems and specifications have been proposed in the literature, each targeting at a specific setting.

We have presented $\mathcal{COLOS}$, a generic model of component lookup, which can be used to express most predating lookup schemes. $\mathcal{COLOS}$ matches component specifications against templates using mathematical formulae. We have described this matching through denotational semantics, illustrated it through various examples, and presented an implementation of $\mathcal{COLOS}$ in Eiffel. The implementation reflects exactly the theory and uses automatic transformations as well as infix operators to obtain extremely compact and intuitive code. We envision the definition of further "common" matching modules, and intend to implement our framework on top of a fully decentralized peer-to-peer overlay network. Furthermore, we plan to port it to a wider range of programming languages and platforms in order to obtain interoperability.

```
uid_specification_term: LL_UID_SPECIFICATION_TERM
regexp_specification_term: LL_REGEXP_SPECIFICATION
load_specification_term: LL_LOAD_SPECIFICATION
...
create uid_specification_term.make ("1337")
create regexp_specification_term.make ("This component...
                author: John Doe")
create load_specification_term.make (Current.component)

Current.add_specification_term_to_spec(
   uid_specification_term)
Current.add_specification_term_to_spec(
   regexp_specification_term)
Current.add_specification_term_to_spec(
   load_specification_term<10.0)
...
```

Figure 5.2: Specification declaration

```
uid_template: LL_UID_TEMPLATE
regexp_template: LL_REGEXP_TEMPLATE
load_template: LL_LOAD_TEMPLATE
component: LL_COMPONENT
components: LL_COMPONENT_COLLECTION
...
create uid_template.makr ("1337")
create regexp_template.make ("*author: John Doe*")
create load_template.make
component:= (components^((uid_template+regexp_template)
   /(1.0+load_template))).get_first_component
...
```

Figure 5.3: Using the lookup infrastructure

Chapter 6

Using Origo

In previous Chapters Origo Core and the development platform built using it are described. To make the platform usable as a central point for managing project life, we provide a web interface. It should provide all necessary functionality to manage a project hosted on the platform. It uses and extends the use cases defined in Origo Core. Additionally a work item system is integrated into the web interface. Work items display notifications about changes and evolution of a project on the Origo-Home page. Work items can be used by a developer or a user to keep track of changes in his own and bookmarked projects.

6.1 Design

The web interface uses the free open-source content management system Drupal [38]. Drupal provides an extension system with themes, modules and hooks. It has a big user community and therefore many extending modules and a good documentation exist. Drupal provides powerful tools like an API for web forms which allows the creation of forms that can be designed using the theme system.

The Origo theme for Drupal is built using this themeing framework. It uses PHP files defining the structure and CSS files defining the rendering of the page. Drupal can be extended with modules. These modules use hooks to interact with internal Drupal processes. Relying on this extension mechanism, it is normally not necessary to modify code in Drupal Core itself (for the web interface of the development platform there is one exception, described later).

Figure 6.1: Work item icons

6.1.1 Work items

Work items are notification on changes in a project. They help a developer to keep track of project evolution.Work items can be used by other platform users who want to stay informed about the development of projects they are interested in. Work items are created on key events in the system. For example when a wiki page is edited or Subversion commits takes place. Some work items are created and triggered in the web interface others come from Origo Core use cases and commit work items are triggered by Subversion server hooks. Each work item has its own icon, seen in Figure 6.1.

Work items are stored and managed within the central database of the development platform and are therefore not restraint to a specific project page. This way a user can access all relevant work items, no matter what project site he is currently on. Several use cases are available to create, view and manage work items. To distinguish between new and old work items a work item is marked read when a user reads the work item. For implementation details see Section 6.7.

6.1.2 Drupal sites

Drupal provides a sites system to host several sites with only one Drupal code base. Each project has its own directory within the *sites* directory and can have its own modules or themes. Modules that available to all projects are stored in *sites/all*. The correct site is determined by looking at the entered URL, if no match is found the default site is displayed. For the development platform, we do not have project specific modules, therefore all additional modules are located inside *sites/all*. To provide better maintainability the site specific settings file is modified to use an include file. This makes it easier to make changes to the settings file. Only the database settings remain in the project specific file.

6.1.3 Scalability

Origo Core is designed to scale. The web interface is also built with this goal in mind. It is possible to distribute the hosted projects pages and corresponding databases across multiple web server machines. To support more projects the databases can be moved to other servers. It is possible to use several database servers because the database server can be set individually for each project site. The authentication and authorization system we have implemented for the web interface can handle logins over several web and/or database servers.

6.2 Drupal modules

6.2.1 Origo Auth: authentication and auhorization

The Origo Auth module handles user registration, log in, log out, password change and password reset. It contains a session handler that scales across multiple machines and a includes an XML-RPC client.

User Registration: When registering a user with the normal Drupal user registration, a user is created in the Drupal database belonging to the project page the registration form was filled out. For the development platform we need a global user registration which creates an Origo user by using XML-RPC in the platform database of the back-end as well for all integrated applications and services.

To achieve this we modified the existing user registration for Drupal sites. Using the Drupal hook system this is possible without changing code in any of the Drupal core modules. First the `hook_form_alter` replaces the validate and submit functions of the registration form. The form now calls our own functions instead of the functions defined in the user module.

The submit function `origo_auth_register_submit` then makes an Origo API call to `internal_user.add` to create a new user. On success the function `origo_auth_authenticate` (see Section 6.2.1) creates the Drupal user and logs him into the system.

Session Handling: Drupal provides a simple session management using PHP session. This system works of course fine for single Drupal instances, but does not meet all requirements for the development platform. For one thing we would like to keep the user logged in not only in the local project. If he goes to another project (ie. another Drupal instance) he is still within

the development platform and should be logged in. One possibility would be to store the PHP session ID in a cookie accessible in all projects. While this is possible for a single server environment, it does not work with multiple servers because the session itself would have to be transfered to the other server. Because of security reasons neither the storage of the session on the client is a solution.

Another problem is that each Drupal instance has its own user database. So if a user is logged in to one project, we cannot simply log him into another project instance because the user might not exist in the other user database. Using just one user database would limit scalability, and keeping them synchronized would be very complicated.

The third problem is that the development platform itself has a session system. If an Origo session expires one has to log in again to get a new session. Therefore we need the username and the password.

The solution we came up with is to store an additional cookie on the client. This cookie stores the Origo username and the encrypted password. With this information available in all Drupal instances we can simply log the user on to every instance using the login function.

The Origo session system is an extended Drupal session system using PHP sessions and the additional cookie. The green part in Figure 6.2 shows the session system in a flowchart. When accessing a page the system checks if the Drupal cookie and/or the Origo cookie is available.

- If both cookies are missing the user is just an anonymous user.
- If the Drupal cookie is missing and we only have the Origo cookie a login using `_sess_load_origo_user()` is performed.
- If the Drupal cookie is available and the Origo cookie is missing we remove the PHP session and log out the user in this instance. This means deleting the Origo Cookie performs a logout on all projects.
- If both cookies are available we check if they are valid and both for the same user. If so a login using the PHP session and the normal Drupal session system is performed. If the user data does not match a login using the Origo session and `_sess_load_origo_user()` is performed.

The function `_sess_load_origo_user()` extracts the user and the encrypted password from the cookie. After decrypting the password `internal_user.login` is called using XML-RPC to log in the user into Origo and get the Origo session. The Origo session is stored in the Drupal user object. After the Origo login the user has to be logged in into Drupal. A

check on the local user database shows if the user is already available. If so, his data is updated, otherwise he is added to the database. To assign the correct access rights it is then determined if the user is an administrator, a project owner, a project member or just a normal Origo user. See the orange part in Figure 6.2 for a graphical representation of the function `_sess_load_origo_user()`.

Because at the time the session code is executed most of the Drupal code is not yet loaded, we cannot use the integrated `xmlrpc()` function. We use the functions from the PEAR package XML_RPC [87] instead.

User log in To intercept the Drupal login system the form validate callback for the login form is changed using the hook `hook_form_alter`. We use the validate handler instead of the submit handler in this case because the XML-RPC request validates the entered data and because we do not want to overwrite the rest of the login performed in the submit handler. The new validate handler calls `origo_auth_authenticate` function to perform the login on Origo. First a XML-RPC request to `internal_user.login` is executed. This call returns the Origo session which is stored in the Drupal user session. After the successful call the role of the user is determined using `authorization.is_allowed_project` XML-RPC requests. Finally the Origo cookie containing the username and encrypted password is created.

User logout As described in Section 6.2.1 the complete logout in all project instances is performed by destroying the Origo cookie. Therefore we use `hook_user` to hook into the user logout and destroy the Origo session.

Password change: The Origo Auth module overwrites the Drupal password change. The existing password change would only change the password in the project specific Drupal database. For Origo we need to change the password directly in Origo using a XML-RPC request. Drupal has a password change field on the user edit page. This page also has some fields that cannot be used with Origo, so we use the hook `hook_menu` to intercept the original edit menu and create a new one currently only containing the fields that allow changing the password. The corresponding submit function sets a new password calling `internal_user.change_password` over XML-RPC.

Lost password: If the user forgets his password he needs a way to reset it. As for the user registration (see Section 6.2.1) we use the hook `hook_form_alter` to change the form validate and submit handlers of the existing password reset function. The new submit handler starts an

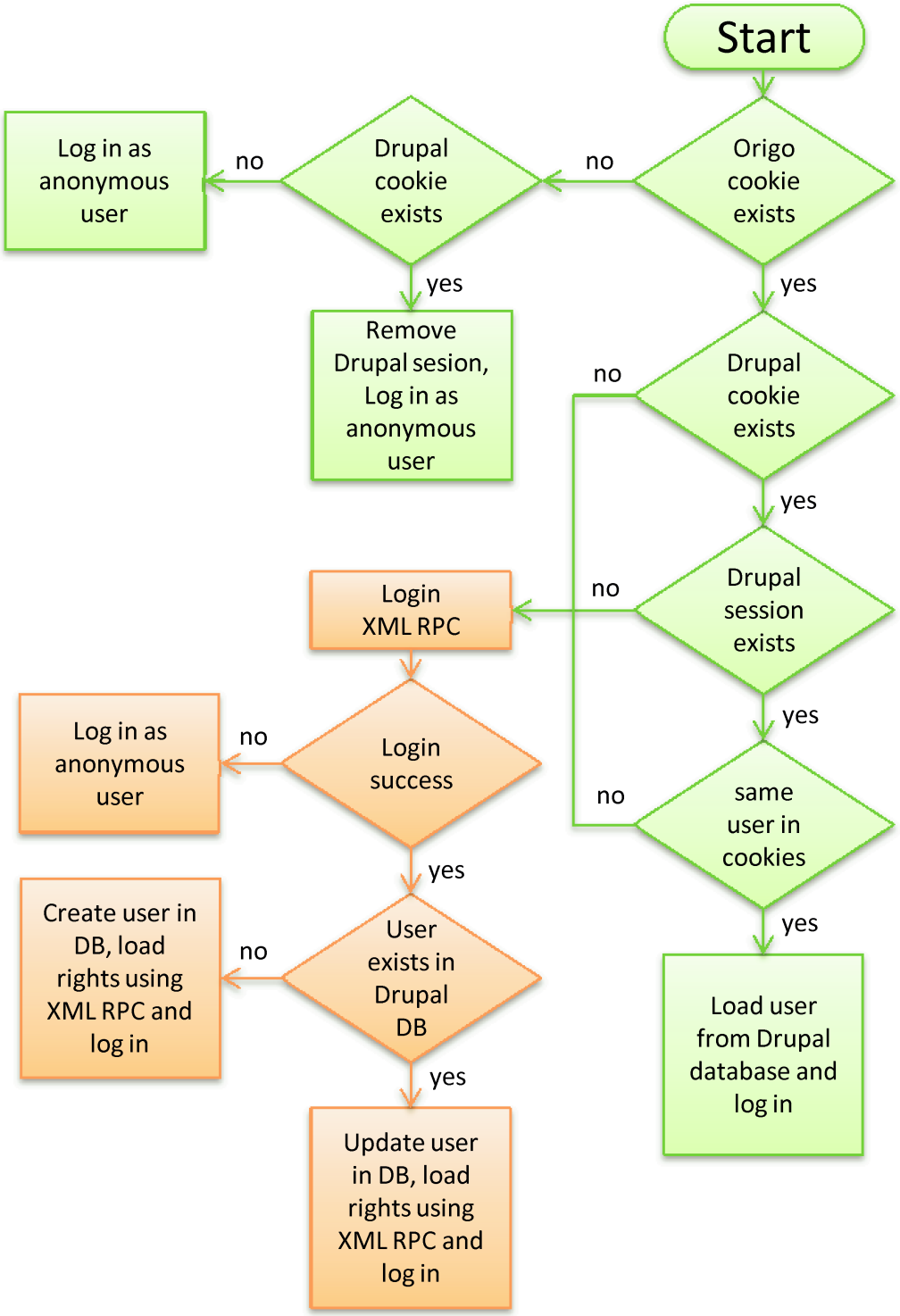


Figure 6.2: Session handling and user log in

XML-RPC request to `internal_user.reset_password`. This call starts the `USER_RESET_PASSWORD` use case in Origo Core which takes care of generating a new password and sending it to the user in an email. This generates also the password for all the external applications like SVN and FTP that the user belongs to.

XML-RPC wrapper: Many of the Origo API calls require a valid session. This session is returned by the `internal_user.login` XML-RPC request and is only valid for a limited amount of time. If the session expires each request that requires a session will return an error. A relogin with `internal_user.login` is necessary to get a new session. To provide an automated relogin if a session is expired the Origo Auth module provides a wrapper to the Drupal `xmlrpc()` function.

There are two functions available: `origo_auth_xmlrpc()` which basically is the same as the original `xmlrpc()` and `origo_auth_xmlrpc_session()` which is used for API calls that require a session.

`origo_auth_xmlrpc_session()` adds the current Origo session as first argument and calls the Drupal `xmlrpc()` function. If this function returns an error indication the session is not valid, a relogin using `_sess_load_origo_user()` (see Section 6.2.1) is done. After the relogin `xmlrpc()` is executed again with the new session. If there is still an error it has to be a serious problem and the error is therefore given to the caller.

6.3 Origo-Home

The Module Origo-Home is the Origo web main module and provides besides several other functions features for work items, project settings, releases and Origo administration. Functions are organized in groups and moved to include files whenever it was possible. All functions are still defined in the hook `hook_menu` which defines the path a function is available at.

Origo-Home is the main page to view work items. A project tab is shown for each project a user is either developer or has bookmarked it. Normally only the unread work items are shown and they become read by following the link or using the checkbox. When selected to show all work items the read workitems are shown too.

This page makes several XML-RPC requests to Origo Core. First the own and the bookmarked projects are retrieved using `project.list_of_user` and `user.list_bookmark`. Wit `workitem.list_projects` the workitems for these projects are retrieved (see Section 6.7.8) and listed in a table for each

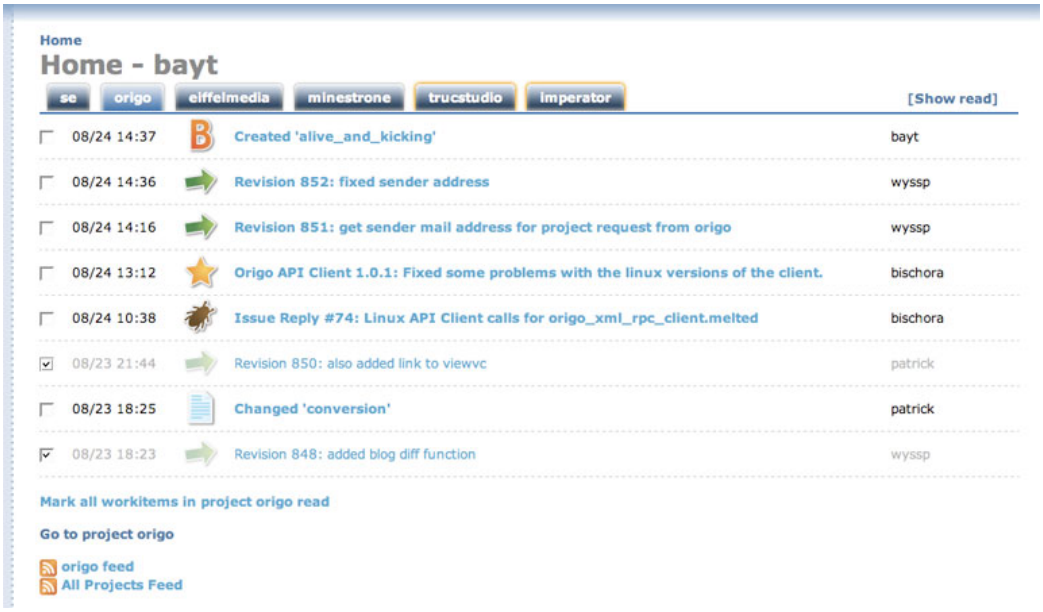


Figure 6.3: Origo-Home showing the work items

project. Depending on the work item type the information shown in the table is extracted and displayed.

To enable fast tab switch a JavaScript is used. Drupal contains the JavaScript library jQuery, which makes it straightforward to add fancy effects or AJAX to a page. A JavaScript together with an AJAX request is used to set the read state of a work item. A click on the checkbox fires an asynchronous request which sets the state in Origo using XML-RPC. As soon as this request completes the redering of the work item is changed to look greyed-out or marked unread to look bold and colorful again. Origo-Home has a link to mark all workitems of the project as read it calls `workitem.set_read_status_project`.

Work item subscription: The work item subscription settings provide a simple way to manage the work item notifications (see Section 6.7.7). Using calls to `project.list_of_user` and `user.list_bookmark` the own and the bookmarked projects are retrieved. For each of these projects `user.list_workitem_subscription` retrieves the currently set notifications. After submitting the form an XML-RPC request with `user.set_workitem_subscription` sets the new notifications.

Project bookmarks: Origo-Home module provides a list of all bookmarked projects which it gets by a XML-RPC request to `user.list_bookmark`. There are also two menu paths defined in the hook `hook_menu` to add and remove bookmarks. These paths can be used as links to quickly add or remove a bookmark and are implemented by calling `user.add_bookmark` or `user.remove_bookmark` XML-RPC. Both adding and removing also sets or removes all the workitem subscriptions (see Section 6.7.7) for the corresponding project.

Project list: The project list contains all projects hosted on Origo. This list uses the internal API call `internal_project.list` to retrieve the projects because the list should also be available to anonymous user which do not have session. Projects flagged hidden are not shown in this list. Most of the hidden projects are student projects created in courses at ETH.

User key request: External software using the Origo API requires the user to enter a user key instead of his password. This page provides a way to request a key using the API function `internal_user.generate_key`.

E-Mail change: This page allows a user to change his e-mail address and is implemented using a XML-RPC request to `internal_user.change_email`.

Project settings: On the Project Settings page a project owner can manage some project settings.

The members page allows adding and removing project members or owners. This is done by calling `project.change_group` with XML-RPC.

The description page allows changing the project description using the XML-RPC methods `project.retrieve` and `project.change_description`.

Changing the logo is possible on the logo settings page. Adding a logo uploads the picture to the local Drupal instance and uses the Drupal theme system to display the logo. Additionally the executed XML-RPC method `project.change_logo` updates the logo filename in Origo.

Project creation request: Every Origo user can request the creation of a new project. The project creation itself is done manually by an administrator for security reasons. However several features are implemented to automate this process.

A user can request a project on the create project page. He has to provide the name, a description and if it is a closed or open-source project. An XML-RPC request to `project.request_add` checks if the project name is valid

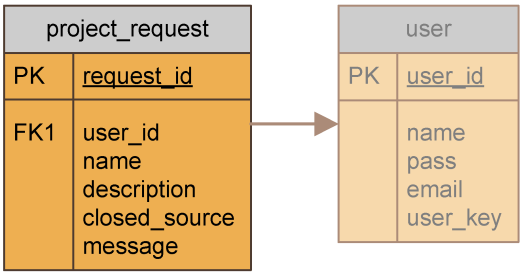


Figure 6.4: Project request table

and still available and adds the project to a request table in Origo (see Figure 6.4). Using the Drupal mail function a mail to the administrators is sent including the entered data and a link to a creation form. This creation form defined in `origo_admin_create_project_form_request_page` is only available to administrators and loads the details for the requested projected with the XML-RPC request to `project.request_retrieve`. The administrator may now make changes to the entered data and the confirmation mail. Sending the form starts the project creation process which includes the project creation, adding the requesting user as project owner (see Section 6.3) and sending a mail to the user informing him about the created project.

Administration menu: The admin menu provides an interface to the XML-RPC methods reserved for administrators.

There is a project list like the one open for all users (see Section 6.3). The difference to the open project list is the usage of the external API call `project.list` and that is also shows the hidden projects.

To send newsletters or important information to all users an administrator can use the mass mail function defined in `origo_admin_massmail_page()` which starts a XML-RPC request to `origo_system.mail_all`.

A project creation form allows the direct creation of a project without using the request mechanism described in Section 6.3.

Finally the function `origo_admin_status_page` shows the result of the XML-RPC request to `origo_system.status` which returns some information about the running nodes.

6.4 Issue tracker

The issue tracker module is the web interface to the Origo issue system. Issues are implemented in Drupal as a new node type and issue replies are simple comments. The hook `hook_insert` is used to intercept node insertion

Figure 6.5: Checkbox to flag a page private

and make a call to `release.add` via XML-RPC. This call returns the project specific issue ID which is stored in an additional table *issues* in the Drupal database. The standard comment system is also modified in the hook `hook_form_alter` to execute the XML-RPC method `release.comment`

6.5 Developer pages

The Developer Pages module is a simple module that adds the possibility to flag pages as private. Private pages can only be accessed by project members and can be used to store project internal information. Work items created from private pages are also only visible to project members.

This module is a Drupal node access module and uses several hooks to perform its task. `hook_node_grants` is used to define the node access rights, `hook_nodeapi` is used to keep track of node inserts and changes to update the table containing all private pages. `hook_form_alter` is used to add a checkbox at the end of a node edit form (see Figure 6.5) giving the possibility to mark this node private.

6.6 Existing modules

Captcha Module: Because spam bots are everywhere nowadays it is necessary to protect all functions that can be accessed without a valid login. This includes user registration and password reset. The Captcha module provides a simple math challenge a user has to answer. The protection is not as strong as it would be with an image captcha, but the image captcha module had several bugs which made it impossible to use. Fortunately at the moment the current system suffices.

Diff module: Diff shows differences between node revisions. It adds a new tab on top of nodes like wiki pages and shows all changed word in a colored view.

Form store module: Provides form information to other modules and is needed by the Captcha Module.

```
wget http://www.google-analytics.com/urchin.js -q -O
/data/www/Origo/static/urchin.js
```

Figure 6.6: Cron job command for Google Analytics

GeSHi filter module A filter to highlight source code using GeSHi. [88]

Google Analytics module The Google Analytics module is used to gather advanced web statistics using Google Analytics [89]. It works by including a JavaScript on top of each page and can therefore retrieve information that is not available in web server logs. The included JavaScript is hosted on `www.google-analytics.com` which turned out to be a bottleneck, therefore we modified the module. Instead of including the script from `www.google-analytics.com` we used a local copy on our local server. A simple daily cron job (see Figure 6.6) is scheduled to download the script to make sure the script is up to date in case Google releases a new version.

Google Co-op CSE module Google Custom Search Engine [90] is a service to include Google search on your on web site. We use this service to provide an Origo wide search over all projects.

Image module: Allows uploading, resizing and viewing of images.

Image assist This module allows users to upload and insert images into posts. It automatically generates an add image link below text fields.

Pathauto module: Provides a mechanism for modules to automatically generate aliases for the content they manage. This is used to generate wiki links.

PEAR Wiki filter: Filter which uses the PEAR Text_Wiki [91] package for formatting.

Tag Query Language: A nice tag query language. This can be used to write queries to retrieve nodes with specific tag. For example one could to write a query for all open issues assigned to him.

Wikitools module: Provides helper functionality to have wiki-like behavior.

6.7 Work item implementation

workitem_id The work item ID, unique in the system

type The work item type (1=Issue, 2=Release, 3=Commit, 4=Wiki, 5=Blog)

creation_time Time stamp when the work item was created

project_id ID of the project this work item belongs to

project Name of the project this work item belongs to

user Name of the user responsible for the work item creation

is_read 1 if the user has already read this work item, 0 otherwise

6.7.1 Issue work item

Issue work items are created for new issues and issue replies. The following additional information is included:

project_issue_id The issue ID, unique in the corresponding project

title The title of the issue

description Detailed description or text provided in the issue

is_new 1 if this is a new issue, 0 if it's a reply

url Link to the issue web page

Origo API call: Issue workitems are created after inserting the issue itself in the `ISSUE_ADD` and `ISSUE_COMMENT` use cases, which are started by the XML-RPC methods `issue.add` and `issue.comment`. Because issues are already stored in Origo only the `issue_revision_id` needs to be stored in the table `workitem_issue`. Figure 6.7 shows this relation.

Drupal integration: The issue work items are created on new issues and issue comments in the Issue Tracker module. (see Section 6.4)

6.7.2 Release work item

A release work item is created on each new release. It contains the following information:

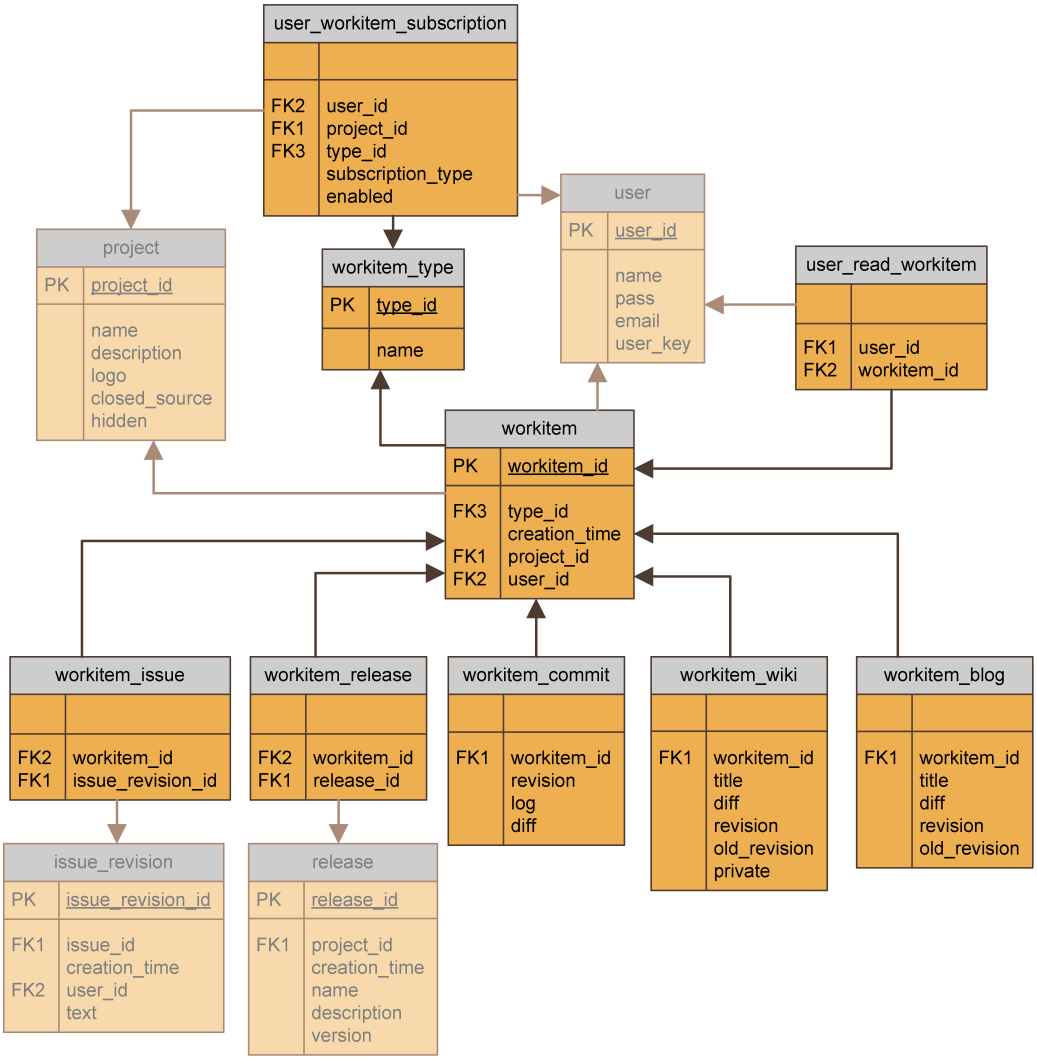


Figure 6.7: Work item tables

name The name of this release

description Detailed description

version Version of this release

url Link to the download page

file_count Number of files included in this release

file_name_X Filename of file X (X = {1 .. file_count})

file_platform_X Platform for file X

Origo API Call: When adding a release with the `release.add` API call Origo starts the `RELEASE_ADD` use case. After inserting the release itself it takes the ID of this releases and inserts a new workitem into the tables `workitem` and `workitem_release` as shown in Figure 6.7.

Drupal integration: When using the web site for releasing files the work item (and release) creation is triggered inside `origo_home_create_release_form_submit` in the `origo_home` module.

6.7.3 Commit work item

A commit work item is created for each commit in the Subversion repository. It contains the following information:

revision The SVN revision associated with this commit

log Log describing the commit

url Link to the WebSVN page for this revision

diff Diff for committed files (this is truncated for large commits)

Origo API call: Commit work items are created in the internal XML-RPC method `internal_commit.add` which starts the `COMMIT_ADD` use case. As shown in Figure 6.7 all data is stored in the table `workitem_issue`.

Subversion integration: The commit work item creation is triggered by a SVN post-commit hook. The used script is an adaptation of the standard commit mail script which uses XML-RPC instead of mailing the changes. The script gets the user, project, revision, commit log and generates a diff of all changes which are then used to call `internal_commit.add`.

6.7.4 Wiki work item

Wiki work items are created for new and changed wiki pages and contain the following information:

title Wiki page title

diff Diff of wiki changes

revision Drupal node revision after change

old_revision Drupal revision before change

url Link to the wiki page

diffurl Link to the diff page for this wiki page

Origo API call: The use case `WIKI_ADD` started in the XML-RPC method `internal_wiki.add` adds wiki work items into the `workitem_wiki` table (see Figure 6.7). Besides a diff between the revisions which is generated using the PEAR Text_Diff class [92] the old and new node revision is stored.

Drupal integration Adding or editing wiki node types is intercepted in the Origo-Home module (see Section 6.3) within the hook `hook_nodeapi`.

6.7.5 Blog work item

Blog work items are created for new and changed blog posts and contain the following information:

title Blog title

diff Diff of blog changes

revision Drupal node revision after change

old_revision Drupal revision before change

url Link to the blog entry

diffurl Link to the diff page for this blog entry

Origo API Call: Adding blog work items in Origo is similar to adding wiki work items. It uses the API call `internal_blog.add` to store blog work items in the table `workitem_blog`.

Drupal integration: Adding or editing a blog node type is intercepted with the hook `hook_nodeapi` like for wiki workitems.

6.7.6 Access Control

While blog and release work items are accessible for everyone, commit, wiki and issue items have an access control mechanism. Commit items for closed-source projects are of course only visible for project developers to keep the source closed. Wiki pages and issues can be set private so we have to do the same with their work items. If the corresponding wiki page or the issue is private the work item is only visible for project developers.

6.7.7 Notification

Origo provides different ways to notify users about new work items. First there is the work item list on Origo-Home. New work items can also be queried via the API and mail notification is available.

A user can set how and for which work item types he wants to get notified. This is done using the web interface (see Section 6.3) or directly using the API. The XML-RPC method to be used is `user.set_workitem_subscription`. The core use case `USER_SET_WORKITEM_SUBSCRIPTION` then adds the subscriptions into the table `user_workitem_subscription`. There is also a call `user.list_workitem_subscription` available to read out the current subscriptions.

6.7.8 Work item retrieval

To retrieve the work items there is either the XML-RPC method `workitem.list` or `workitem.list_projects`. The work item list on Origo-Home is implemented with a call to `workitem.list_projects` which retrieves a given number of the newest work items for each own and bookmarked project. The parameter `unread_only` is used to retrieve only unread work items, otherwise read and unread will be retrieved. The method `workitem.list` also lists a given number n of work items, but this method just returns a total maximum of n newest work items in all own and bookmarked projects. If five work items are requested and the first project already has ten new work items then only five work items of this project are retrieved and none from any other project.

A detailed single work item is retrieved with a call to the XML-RPC method `workitem.retrieve`.

All retrieve calls implement the access control described in Section 6.7.6 and the subscription settings described in Section 6.3. So only work items a user wants to see and is allowed to see are retrieved.

6.8 Teaching

Our work involves teaching software engineering and software architecture to large numbers of students. The teaching activity is always accompanied with practical work in software projects. First year students learn programming using the method of the inverted curriculum [93] where they acquire programming skills by using existing libraries first as clients and then start extending them. In higher level courses we encourage students to participate in existing open-source projects to practice distributed software development and interaction with teams consisting of globally spread engineers. In both scenarios Origo is the information management platform that students use to coordinate their efforts. Students can create projects hosted on Origo and use them as communication platform. To prevent that immature or unfinished projects clutter the platform, such student projects hosted on Origo can be hidden from the publicly available projects list.

In Section 6.8.1, we present an evaluation on how working in open-source projects affects student motivation.

6.8.1 Open-source projects in programming courses

We teach software engineering techniques, design patterns, and project development models in our courses. Teaching these topics to students without actually exposing them to industry grade software may not only sound preachy, but may also leave no remaining impression. Open-source software offers great opportunities to bring real-life experience directly into the classroom [94, 95]. In particular, open-source software can be used to emphasize the importance of high quality software design, the role of design patterns, the need of good documentation, and the relevance of social skills in a real-world environment.

Over the last years, only a few instructors reported on their experimentation with open-source software in the classroom. Allen et. al. [96] used Dr. Java - an open-source Java programming environment - to teach extreme programming techniques. Students of Fuhrman [97] freely chose an open-source project which they inspected to propose corrections in the design. Fuhrman did not require students to implement their improvements, but many still did. Carrington [98] allowed students to choose from a list of open-source projects that are useful to software engineering and let them inspect, report, and extend the tools during his course. Except for contributions by students to Dr. Java, these experiments refrained from actively submitting changes, bug fixes, or improved designs to the open-source projects.

The assignment described here combines the freedom of choosing an ar-

bitrary open-source project with the ultimate goal of students contributing to it. Allowing students to freely choose the project, on which they intend to work, lets them adapt the assignment to their personal interests. Requiring students to contribute back to the open-source project, increases the prestige of their work and the effort that they are willing to put in the assignment. Students experience all the tasks needed to adapt existing software: understanding it, identifying a needed improvement, designing the solution, implementing, testing, and adapting to the requirements for contribution. During this work, they need to get socially involved with the other developers of the open-source project, having to deal with the communication problems that occur frequently with distributed development.

This work presents a thorough evaluation of the approach by using a questionnaire focusing on students' motivation [99]. The study compares the motivation of students working on an open-source project to the motivation of a control group from another course working on a traditiona(exercise) project.

Course setup: This study focuses on two programming courses: an advanced Java course (the **experimental course**, roughly 70 students, master-level) and a course on concurrent object-oriented programming (the **control course**, roughly 30 students, master-level). Both courses are elective courses that can be chosen both by Bachelor and Master students, but also by post diploma students (e.g. PhD students). For the project the experimental course required groups of students (maximum 5 people) to contribute to a Java open-source project of their choice. The control course relied on a given artificial project to be solved by groups of maximum 3 people. The goal of the projects was in both cases to deepen the understanding of the lectures by putting the concepts to work. Both projects spanned over 6 weeks and started at the middle of the semester. For both courses the results of the project influenced significantly the final grade of the course (respectively 40% and 65% of the final grade).

The following sections describe each of the project assignments in more details.

Control course project: In line with the focus of the course the project required the students to build a group of command line programs, that use a given concurrency framework and demonstrated its capabilities. The second part of the project consisted in extending the framework with a rendezvous synchronization mechanism that did not exist previously.

The exact assignment was to: (1) program the required tasks, (2) answer

questions, and (3) write a report including a description of the code as well as the answers to the questions.

Open-source project: At the beginning of the course, students were informed about the grading scheme and that they would have to form groups for the project. Students received the project description only at the middle of the semester. At this occasion, lecturers and students discussed the nature of the community work that open-source projects involve. The discussion showed that very few of the seventy students had actively participated or even contributed to an open-source project. Obviously, all of them had already used open-source software.

The project description included an initial selection of popular and active open-source projects (see Table 6.1). The main criteria for selecting these projects were activity of the community, existence of a project plan, existence of a bug tracking system, and the availability of the code through a repository so that students are able to create branches on which they can work.

Table 6.1: Open-source projects

Projects Suggested		Anteater: http://aft.sourceforge.net ArgoUML: http://argouml.tigris.org JEiffel: http://se.ethz.ch/projects/benno_baumgartner JSR's: http://www.jcp.org Open Office: http://www.openoffice.org Tomcat: http://tomcat.apache.org XmlIO: http://www.bifrost.org/xmlio/index.shtml
	Projects Chosen	Azureus: http://azureus.sourceforge.net Eclipse: http://www.eclipse.org GPSylon: http://www.tegmento.org/gpsylon JUnit: http://www.junit.org/index.htm Maven: http://maven.apache.org
Other Projects		CaCMS: http://cacms.sf.net Columba: http://www.columbamail.org FreeGuide TV: http://freeguide-tv.sourceforge.net Gham: http://www.hattrickitalia.org/gham Hunt for Gold: http://huntforgold.sourceforge.net JackSum: http://www.jonelo.de/java/jacksum Jython: http://sourceforge.net/projects/jython Tapestry: http://tapestry.apache.org WTflash: http://sourceforge.net/projects/wtflash

Each description of the proposed projects included references to the relevant web pages, wikis, mailing lists, and URLs of the bug tracking systems. It also included an informal evaluation of the projects' organization such as specifics about the development process (e.g. was it bug-driven or planned). This information provided a starting point when looking for possible contributions to the project.

Besides this selection of projects students could also look for other Java open-source projects and assess their suitability for contributions. As Table 6.1 shows, half of the students chose one of the proposed projects and the other half selected one of their own.

The assignment was to: (1) get an overview of the project, (2) identify the parts to which they would contribute code, (3) contribute, (4) write a report recalling their experience.

Outcomes: The students generally spent a lot of time and produced quality software.Their contributions include the world map plug-in integrated in the latest versions of Azureus, various bugfixes for the Eclipse Maven plug-in, CSV-exporting Tapestry components, bugfixes and extension of the GPL Hattrick Manager, bugfixes of JUnit, bugfixes and major improvements of the Columba Mail Client,extensions of GPSylon, bugfixes of the FreeGuide TV, a GUI for JackSum, extensions of WTFlash, extensions to the game "Hunt for Gold", and finally extensions to CaCMS to include WebDav support. These are only the projects producing code that is shipping in the products. The course's wiki page[1] provides much more details.

6.8.2 Evaluation of motivation

Background: Students who are motivated by a task are more likely to succeed in solving it. According to Pintrich, students' motivation is strongly correlated with their academic success [100]. In particular, Pintrich shows that students are more motivated if:

1. they believe that they are able to solve the task at hand.

2. they feel in control of their learning.

3. they are personally or situationally interested in the task.[2]

[1] http://wiki.se.inf.ethz.ch/tjp_06/index.php/Project_page

[2] Personal interest describes a disposition of an individual to be attracted to a particular activity or topic while situational interest describes a state of an individual where the interest results from the task itself.

4. they believe what they are doing is valuable and useful for themselves.

5. they have social and/or academic goals that they pursue.

By comparing the above motivational influences for the open-source project approach and the traditional project approach, one can expect the open-source project to increase personal and situational interest (3) and to give them the impression that the task is more valuable and useful (4) than the traditional projects'. One can also expect that the traditional project approach is better at providing assurance of success (1) and impression of control (2).

To assess the validity of using an open-source project against using a traditional project, a good indicator is to compare the motivation associated to both projects. The Questionnaire on Current Motivation (QCM) [99] assesses the current motivation of students working on a specific task and therefore is well suited for such a purpose. The QCM is based on the "classical" model of motivational psychology [99] (see Figure 6.8) that states that personal and situational factors influence the current motivation which in turn influences behavior i.e. learning. It also benefits from a whole theoretical psychology framework that allows to easily compare and aggregate questions.

The QCM itself uses 18 items (questions) that measure four factors of current motivation: *anxiety* (fear of failure), *probability of success*, *interest*, and *challenge*. The full questionnaire is available online[3] and reproduced in the Appendix. The QCM items formulate statements for which students assign 1 to 7 points depending on how much they agree with the statements (1: totally disagree, 7: totally agree). As an example:

I enjoy problem solving tasks like the ones that emerge in the project work.
☐ *1* ☐ *2* ☐ *3* ☐ *4* ☐ *5* ☐ *6* ☐ *7*

Results: To assess the motivational implications of the open-source project assignment, the students of the Java programming course filled in the QCM twice: once just after receiving the project description, and the second time at the end of the project. The control course had a similar setting. In the rest of the section we use the following abbreviations:

QCM$_{Et1}$: questionnaires of the experimental group at the beginning of the open-source project

[3] http://se.ethz.ch/people/pedroni/qcm.html

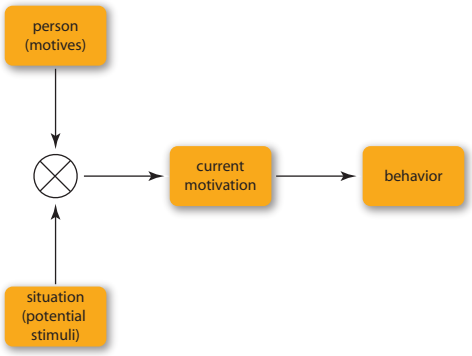


Figure 6.8: Basic model of classical motivational psychology [6]

QCM$_{Et2}$: questionnaires of the experimental group at the end of the open-source project

QCM$_{Ct1}$: questionnaires of the control group at the beginning of the project

QCM$_{Ct2}$: questionnaires of the control group at the end of the project

The responses QCM$_{Et1}$, QCM$_{Et2}$, QCM$_{Ct1}$, and QCM$_{Ct2}$ can be used to compare the two groups in two dimensions:

- Motivational differences between the experimental and the control group (comparing QCM$_{Et1}$ to QCM$_{Ct1}$ and QCM$_{Et2}$ to QCM$_{Ct2}$).
- Motivational changes over time for both of the groups (comparing QCM$_{Et1}$ to QCM$_{Et2}$ and QCM$_{Ct1}$ to QCM$_{Ct2}$).

Differences at the beginning: The 18 items of the QCM were combined into measures for each of the four dimensions. Based on this data, the analysis uses T-Tests[4] for independent sets to obtain the factors that differ significantly between the two groups for each of the two points in time (means are gathered in Figure 6.9).

The comparison shows that QCM$_{Et1}$ and QCM$_{Ct1}$ differ significantly for the factors *probability of success* and *anxiety* (see (∗a) respectively (∗b) in Figure 6.9). The interpretation of such a result is that when students begin to work on an assignment requiring contributions to a real-world open-source project, they feel more uncertain about their success and therefore their fear

[4]T-Tests allow to determine if the means of data differ significantly. In general, this is assumed to be the case if the calculated value $p < 0.05$.

of failure is higher than for a traditional small “toy" project. More surprisingly, students are at this point in time not significantly more interested or more challenged by the open-source project than by the traditional project.

Differences at the end: The comparison of QCM_{Et2} and QCM_{Ct2} shows that the factors *probability of success* and *anxiety* do not differ significantly between the experimental and the control group any more. Interestingly enough, the level of confidence is identical in both groups. The factor *interest* changed (with $p = 0.065$): the interest of the experimental group in their project grew while the interest of the control group diminished. The interpretation is that the fascination of working on an open-source project settles in only after the first hurdle of basic understanding and involvement. Working on a traditional project is interesting in the initial design phase but looses fascination over time.

Another interesting outcome can be detected by comparing individual statements. First, the statement *“If I succeed with the project, I will feel a little proud of my proficiency”* was significantly higher at the end of the projects for the experimental group than for the control group. Second, the statement *“I would also work on a project like that in my free time”* also produced a significantly higher result for the experimental group. This is consistent with the following intuitive interpretation. Contributing to a real-life project is very likely to make students proud and may even make them wish to continue contributing after the mandatory work is completed. But also having completed a project specifically designed for a course is not that rewarding on its own.

Comparison over time for both groups: The second part of the evaluation was done using T-Tests for paired sets of data to obtain significant changes over time. These tests show that for the experimental group the values for the factors *interest*, *anxiety*, and *probability of success* significantly improved (see (∗c), (∗d), and (∗e) in Figure 6.9). It seems that students working on the open-source project first underestimate their capabilities and then gain confidence. This results in a significant increase of the *probability of success* and a reduction of the *anxiety*. The increase of *interest* is probably due to students beginning to understand the challenging and interesting sides of their programming project while working on the open-source project. For the control group no significant change occurred during the project.

Outcome: The open-source project results in a somewhat more unstable situation. Students start with a higher level of fear of failure and a lower level

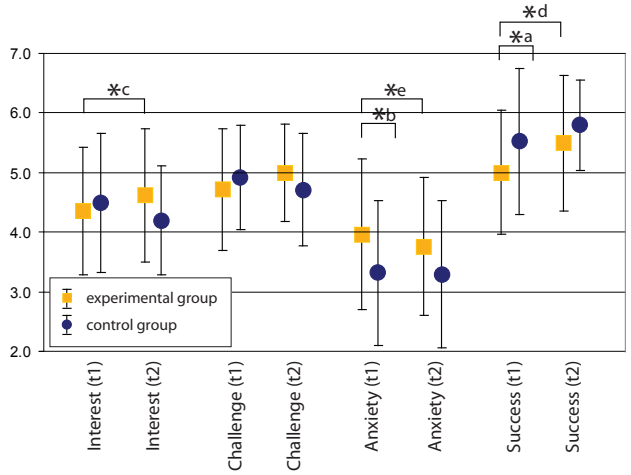


Figure 6.9: Mean and standard deviation of the four factors with a range of possible values from 1.0 to 7.0. (*) denotes significant differences ($p < 0.05$).

of probability of success, while gradually gaining more confidence and finally showing a deeper interest in the subject. In particular, students feel more proud of completing the open-source project and are more likely to deepen their knowledge by continuing to work on it after they finished the official part of the work. This conclusion verifies the assumptions from section 6.8.2 which stated that students working on the open-source project value their work more, but are less confident in their capabilities and control. With the present data it is not possible to declare one of the approaches definitely better than another, but using open-source projects is as good as using traditional projects and helps students build self-esteem.

6.8.3 Complementary items

The previous section showed how an open-source project impacts on students' motivation. To estimate students' activity, learning effect, and commitment additional items complement the QCM. These questionnaires were distributed to the students of the experimental course at three occasions during the course period - at the beginning, in the middle, and at the end.

From over thirty additional items covering questions about the course, the weekly assignments, and the project, three are detailed here. These three items all concern the project and represent a general trend emerging from these additional questions on the project[5].

[5]For details see http://se.ethz.ch/people/pedroni/qcm.html

The first item assesses *activity*. This item addresses how much time is spent on the open-source project. The second item - *learning effect* - quantifies the students' perception on the knowledge learned because of the open-source project. The third item is the *commitment* that results from the project work. This third item was evaluated only at the middle and at the end of the course. The results of these complementary three items help to identify improvements for a future course.

Results: The item addressing the *activity* was: "I am putting much effort into the project". Students assigned 1 to 5 points capturing how much they agree with this statement (1: totally disagree, 5: totally agree). Figure 6.10 shows that the mean effort invested in the open-source project drops towards the middle of the project's lifespan.

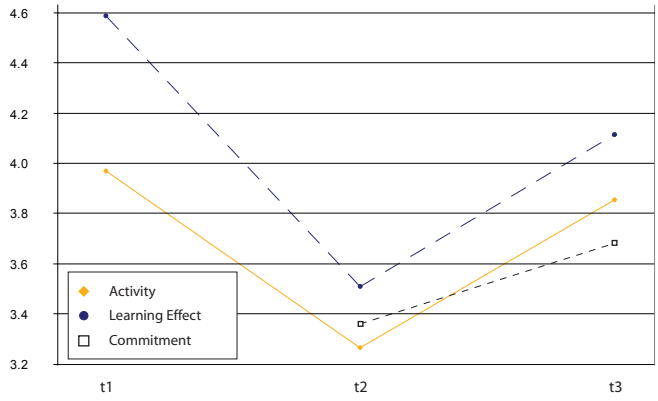


Figure 6.10: Means of *activity*, *learning effect* and *commitment*

The statement "I am learning much by doing the project" assessed the *learning effect* that students attribute to the open-source project. As for the *activity*, the felt *learning effect* decreases in the middle of the course and rises again towards the end (see Figure 6.10).

The item to measure the *commitment* of students was included only twice: at the middle and at the end of the course. It stated "I feel responsible to make the project a success." and the resulting averages show an increase towards the end of the course (see Figure 6.10). Additional comments made by students in the questionnaires support this observation: "*It was a lot of work, but very cool to have taken part in an open-source project.*"

The evaluation for the three additional items used a variance analysis with repeated measures to find out whether the averages between points in time differ significantly. This was the case for all of the presented items.

Outcome: At first sight, the decrease of *learning effect* and *activity* in the middle of the course might be surprising. But it is important to see the course and the project in the context of the other activities the students take part in. In fact, students were busy with midterm exams for other courses at this point in time. To overcome the problem of multiplied pressures, instructors need to consider both the project phases and the other obligations students have at university. In particular, during the time consuming phase of design and implementation lectures could be reduced or transformed into interactive lab sessions.

6.8.4 Conclusions and future work

The paper has two main contributions. Firstly, we show that having students collaborate on open-source projects within the frame of a course is interesting and has advantages over using an artificial project. Secondly, using a study backed up by research in psychology enables a scientifically sound evaluation of the approach.

This article described our first attempt at using open-source projects within a course. It showed that students were obviously afraid at first but felt more proud of their achievement in the end. The next iterations of the course will integrate this result and find solutions to cope with students' fear. A first approach is to give them a much more detailed standard operating procedure to get started with their projects. Such an approach could include the following steps: (1) use the open-source software as a tool, (2) explore the code in search for programming patterns, (3) identify weak points or adequate extensions that result in a contribution, and (4) design, implement, and deliver the code. As a second measure, we plan to have several groups of students work on the same project (but not the same subject) so that they constitute a community within the open-source community. This would then show the projects in a more friendly and social way than currently.

We plan to repeat the open-source project and its assessment with undergraduate students to ensure that the results also apply to this population. Our priority is to develop further the methodology assessing teaching experiments by motivation analysis and identifying other settings in which it applies. In particular, we will experiment the impact of having contests, written exercises, programming assignments, in-class exercises, or in-class practical sessions on students motivation.

Small FAM

QCM questions, adapted from [99]:
(I) I enjoy problem solving tasks like the ones that emerge in the project work.

(S) I think I can tackle the difficulties of the tasks involved in the project assignment.
(S) Probably, I will fail solving the project assignment.
(I) What I like about the project assignment is the role of the researcher that discovers connections.
(A) I feel pressure having to perform well solving the project assignment.
(C) This project assignment is a real challenge for me.
(I) After reading the instructions, the project assignment seemed very interesting to me.
(C) I am very curious how well I will do in this project.
(A) I am a bit afraid of being embarrassed by my performance in the project.
(C) I am determined to work very hard for the project.
(I) I enjoy doing the project, I would not need any gratification.
(A) Failing the project assignment would embarrass me.
(S) I believe everyone can succeed in doing the project.
(S) I believe I won't succeed in the project assignment.
(C) If I succeed with the project, I will feel a little proud of my proficiency.
(A) If I think about the project, I am a bit worried.
(I) I would also work on a project like that in my free time.
(A) The requirements of the project work paralyze me.

(C): Challenge
(I): Interest
(S): Probability of success
(A): Anxiety

Chapter 7

The development and use of Origo: Lessons learned

When building a framework to support construction of scalable distributed platforms it is difficult to know where to start. The approach we took was to identify use cases and take them as indicator on the more general requirements the framework should aim at fulfilling. While defining and refining use cases, we carried out an analysis of the state of the art of development platforms. The construction of a novel software development platform was the practical goal driving the conceptual design of the Origo Core framework and the Lookup library. The analysis of other development platforms permitted to refine the found use cases and most importantly to identify platform design and construction attributes that the framework should fulfill.

7.1 Private Alpha - Fall 2006

We believe that a platform can only become better, if it is used by its developers on a daily basis. After all developing our own open-source libraries (e.g. EiffelMedia [101], EiffelStudio and many others) was the impulse for the need of better software development platforms. We started out using several applications to aid software construction in a loosely coupled fashion.

7.2 Private Beta - Spring 2007

After finishing the development of VamPeer and Origo Core, we were able to start integrating all services and applications into the development platform. To broaden the user community in a manageable way, the platform was used in the Software Architecture course at ETH Zurich. The students registered

35 projects and all interested researchers were encouraged to move their source code or publication repositories to Origo.

The development platform was running on two server machines that shared responsibilities. One machine hosted project web pages API nodes and databases of each of the projects, the other server hosted all source code repositories, the overall Origo database and the download mirror for the software releases. During the private beta phase we started using a separate development server that would – upon commit of code – compile the entire Origo platform and execute all unit- and regression tests on the newly compiled platform. This automation permits to identify problems early, before releasing a new version to the life system.

The functionality of this version of the platform is listed below.

- Origo Core capable of running the use cases
- Web interface
- XML-RPC API
- Framework for generic search
- Configuration management using SVN
- File release platform using FTP
- Integration into development environment EiffelStudio

Many show-stopping issues were found during that time thanks to the regular use of Origoby the beta testers.

7.3 Public Beta - Summer 2007

After successful use for an entire course and with a few dedicated beta testers, we decided to open Origo to a bigger audience. The platform had new features and was running on one additional machine; we were able to host an offsite mirror for downloads and had implemented several more features:

- Second version of web interface
- More XML-RPC API calls available
- Integration into two new development environments: VisualStudio and Eclipse

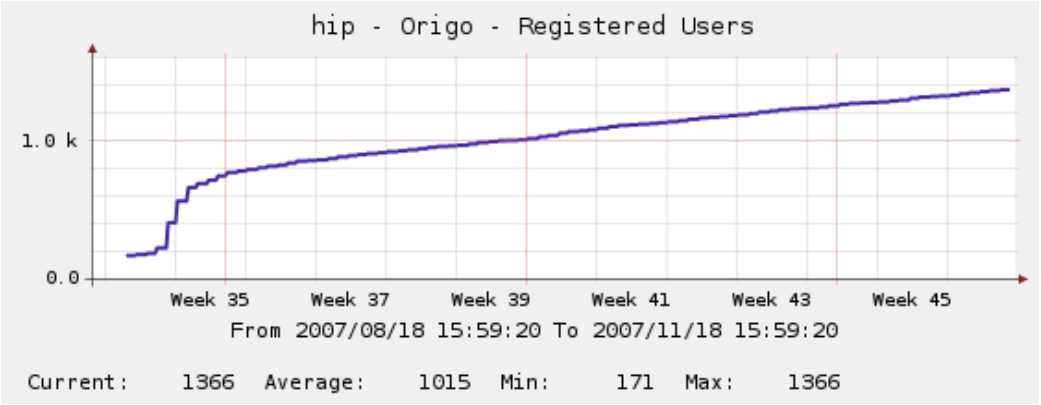


Figure 7.1: Registered Users

Publicizing Origo on the web took considerable effort and we had to work our way up from little known news sites to more and more important ones. On the 25th August a major German computer and IT news site [102] published an article on Origo. From there on things went smoothly and we did not have to make too much effort anymore to be written about. In the week that followed we had to create and register a new Origo project almost every couple of minutes. This overwhelming response has declined, but the growth rate on all aspects of the platform is steady (see next Section for details).

7.4 Metrics

Users: At the time of writing in December 2007 Origo has 1566 registered users. The user base grows with approximately 220 new users per month. On average 23 users are logged on to the platform using the web interface.

Projects: As of December 17th, 2007 - Origo hosts 628 projects out of which 410 are closed-source. Among the hosted projects are many scientific projects that are maintained by researchers just like many commercial projects from all over the world.

Work items: An important measure of activity of all the hosted projects are the work items that Origo manages. The biggest share have the code commits - currently 15'757 commits have been made using Origo. The second most popular action are wiki page edits - they are at 5'393. After that follow

the reported issues with 2'047, comments with 1'054, blogs with 237 and finally the 107 software releases that Origo hosts.

Page accesses: We are using three different methods to collect data on the life web sites of the Origo projects: User behavior on the pages is tracked using Google Analytics as a remote data collection service and with AWStats as a local tracking service. These two are combined with MRTS to track bandwith usage all servers - most importantly the download machines. Since launching the public beta of Origo, the project pages had over 60'000 visitors, accounting to 320'000 hits - the platform is most popular in Europe and Northern America and has server over 340GB of data.

Issues: The development of Origo continues and our active user base reports issues regularly; currently we are 89 open issues out of 238 reported.

7.5 Monitoring and backup

To ensure quality of service, running an online platform for a growing user community requires monitoring many different parts of the platform. As Origo runs on a number of different machines that share responsibilities, we decided to use an off site monitoring tool to track the entire host group. The monitoring software is called Nagios [103] and Mirmon [104]. Miromon tracks only the state of the enlisted download mirrors. Nagios monitors all available services on the different hosts - notably that a host has a connection to a gateway, that a login via SSh is possible, that there is a valid response to a HTTP request and that and FTP login can be made. Nagios can be extended to monitor arbitrary services and therefore correct function of the Origo API is tested as well. Upon failure of any of these services, Nagios sends out alerts to registered Origo developers that can consequently take action and fix the problem.

Security updates on the machines are also sent to Origo administrators and overall the availability so far has been at 99.987%; meaning that the only downtime we had since start of the public availability of the platform was due to restarting the machines after updating kernel images.

Origo hosts a database and a source code repository for every project. Both the database and the repository are not easy to backup just by copying the files elsewhere. The strategy we pursue is to dump both the databases and repositories daily. These dumps are then locally backed up incrementally every day. Additionally the entire contents of all projects and servers is backed up daily to a centralized backup store at ETH Zurich.

7.6 Missing functionalities

The platform targets a light-weight development process. This is reflected in the spartan user-interface support for project management. Currently there is no notion of milestones and no possibility to manage calendars and availability of developers. Users can track time and milestone completion individually by keeping data on the wiki pages of a project but it is desirable to continue development and to find easily manageable analogies that can be leveraged in the user interface.

Besides the mentioned functionalities that are missing, the issue tracker of the platform also contains feature request for several aspects of the platform that bear room for improvement.

7.7 Not desirable functionalities

Among the functionalities that are not desirable is the inclusion of compilation servers. Other development platforms offer compilation farms that allow developers to build binaries of their software. Compilation farms allow creating releases for foreign platforms; not every development team has access to all platforms on which their software runs. In that case a team can use compile servers to create releases for other platforms. The inclusion of compile farms in a development platform poses considerable administrative overhead in maintenance of the operating systems and the hardware. For Origo, we decided to solve this need by offering an API that allows compile servers to register with the platform. This way users can offer compilation services to developers and the administrative overhead lies not with Origo.

Chapter 8

Origo: The vision

8.1 The Impact

Recently, a wide range of services have emerged that support and enable software development, deployment, collaboration and evolution. Each service, however, requires learning curve. Users of a new service must familiarize themselves with the API, create accounts for themselves, and determine how best to integrate the new service into their current workflow. The complexity of individual services and of manually integrating multiple services into a development environment necessarily requires extensive time and effort. Each development team is responsible for setting up such a system if it hopes to provide robust support for its software artifacts and grow a user base that facilitates use and extension of their software. If such development teams and users are to make the most of emerging technologies, they must continuously monitor new and emerging services as they become available. Such a situation is simply untenable because it takes focus away from a teams main task – the development and maintenance of its software.

The Origo Core framework addresses this growing and important problem by providing middleware to facilitate integration and abstraction of a wide range of extant and novel software development services. Moreover, a development platform built on top of Origo Core can be integrated directly into a workflow (graphical IDE or configuration, build, and deployment scripts) so that the complexities of code stewardship are hidden to a development team.

As a result of our proposed design and implementation goals, Origo has the potential for broad impact since it can be used by any development team – may they develop open-source or closed-source software – to simplify their development and deployment processes. Moreover, Origo Core can be used to integrate source, binary, and data repositories like the development platform

that we have built with the framework demonstrates.Through simplification and expedient use of a wide range of current and emerging technologies, Origo and Origo Core help to enable publicity, sharing, extension, and longevity of open- and closed-source software and tools.

At the time of writing, Origo is used by over 1566 users and hosts more than 628 projects. The platform is used by universities and colleges from all over the world as well as by commercial companies and private persons. The platform supports novel teaching directions like integrating collaboration in open-source projects into the curriculum and enables distributed teams from all industries to collaborate in software projects.

8.2 Future work

Future work on Origo focuses on three main goals: Implementing explicit and implicit creation and detection of development communities and to provide specific communication platforms for them. The second goal is to improve the display of projects; the current Listing of projects can take into account activity measurement connected to work items, number of developers on a project and web statistics. For users looking for a certain project on the platform a ranking of the projects can improve the usability. Detecting similarities among projects and suggesting them to users for consideration can be a valuable information source within communities and we plan to offer that. The third goal is to maintain and improve the running platform driven by the reported issues - one popular demand we are considering is the inclusion of a distributed configuration management service.

For the Origo Core framework we want to address some points of the communication infrastructure: We mentioned that VamPeer is incomplete and does not implement the full JXTA specification. Although the current release can be used in Origo Core and in many others applications, it lacks of some essential features that would tremendously improve the VamPeer experience.

We should focus first on a rendezvous server implementation because we then could eliminate the dependency to another JXTA implementation. This task involves working into the rendezvous' peer view protocol. While the rendezvous adaptions are a bigger part, it also involves to enhance the discovery service since this has to fulfill more tasks when the peer is a rendezvous server. We have to maintain an SRDI for all rendezvous clients for example.

Another important issue is the endpoint router completion. It should be able to resolve peer IDs also querying remotely for route advertisements. Another yet unavailable router feature is to forward traffic for other peers

residing behind a firewall.

JXTA is also known for its support to bypass firewalls. The HTTP transport is surely an important contribution to this facility. Such a transport would thus be nice to have.

In JXTA, we generally have secure communication by using TLS channels. As there is no SSL/TLS library for Eiffel yet, it is not that easy to implement the `jxtatls` transport to VamPeer. But wrapping the SSL C library could lead to a TLS transport implementation. The last missing service we would like to list here is the *pipe service*. Its idea is to support virtual channels to one or multiple peers.

8.3 Conclusion

Origo bridges the gap between coding and publication in software development projects. It brings together development teams and offers them an information management platform that is easy to use and integrates thanks to the API and the IDE plug-ins directly into the development process. Accelerating the publication of software releases improves software quality. The overview of work items of all projects facilitates working on multiple projects and with different development teams simultaneously.

The open-source platform Origo allows hosting closed-source projects as well. This makes the platform not only attractive and useful for the classic distributed software development projects known – the open-source projects – but also for all other groups of developers working collaboratively without wanting to disclose their sources.

The Origo Core framework facilitates integration of various applications and services into a one stop platform addressing knowledge management and distribution needs of a given community in a scalable and extendible way. The complexity of making use of extant tools and services for software stewardship and of integrating them into a development workflow is addressed by the framework an it has stood the test of reality and the ever growing user base reassures this claim.

Bibliography

[1] *Sourceforge*. http://sourceforge.net.

[2] S. Oaks and L. Gong. *Jxta in a Nutshell*. O'Reilly & Associates, Inc., Reading, Massachusetts, 2002.

[3] *Google Custom Search Engine*. http://google.com/coop/cse.

[4] T.G. Bay, P. Eugster, and M. Oriol. Generic Component Lookup. *Lecture Notes in Computer Science*, 4063:182–197, June 2006.

[5] *VamPeer*. http://vampeer.origo.ethz.ch.

[6] *XML-RPC Internet Remote Procedure Call*. http://www.xmlrpc.com/spec.

[7] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *Lecture Notes in Computer Science*, 2218:329–??, 2001.

[8] Sun Microsystems. JINI Connection Technology, 1999.

[9] N. Carriero and D. Gelernter. Applications Experience with Linda. *ACM Sympos. on Parallel Programming*, July 1985.

[10] Lindsay Bradford, Stephen Milliner, and Marlon Dumas. Experience Using a Coordination-Based Architecture for Adaptive Web Content Provision. In *COORDINATION*, pages 140–156. Springer, 2005.

[11] K. Lee, A. LaMarca, and C. Chambers. HydroJ: object-oriented pattern matching for evolvable distributed systems. In *OOPSLA '03: Proceedings of the 18th annual ACM Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 205–223, 2003.

[12] M. Oriol and G. Di Marzo Serugendo. A Disconnected Service Architecture for Unanticipated Run-time Evo ion of Code. *IEE Proceedings-Software, Special Issue on Unanticipated Software Evolution*, 151:95–107, April 2004.

[13] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in Java. *Softw, Pract. Exper*, 36:1257–1284, 2006.

[14] S. Sadou, G. Koscielny, and H. Mili. Abstracting Services in a Heterogeneous Environment. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, pages 141–159. Springer, 2001.

[15] Rajesh Krishna Balan, Maria Ebling, Paul Castro, and Archan Misra. Matrix: Adaptive Middleware for Distributed Multiplayer Games. In *Middleware*, pages 390–400. Springer, 2005.

[16] Object Management Group. *The Common Object Request Broker Architecture: Core Specification, Version 3.0.3*. OMG, 2004.

[17] *Simple Object Access Protocol*. http://www.w3.org/TR/SOAP.

[18] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. *http://www.w3.org/TR/wsdl*.

[19] *Sourceforge Enterprise*. http://sf.net/powerbar/sfee.

[20] *GForge*. http://gforge.org.

[21] *BerliOS*. http://www.berlios.de.

[22] *Savannah*. http://savannah.gnu.org.

[23] *Savane*. https://gna.org/projects/savane.

[24] *LCG Savannah*. https://savannah.cern.ch.

[25] *Google Code*. http://code.google.com.

[26] *SourceFubar*. http://www.sourcefubar.net.

[27] *Codehouse*. http://codehaus.org.

[28] *Opensymphony*. http://www.opensymphony.com.

[29] *Javanet*. http://java.net.

[30] *Tigris*. http://www.tigris.org.

[31] *PicoForge*. https://picoforge.int-evry.fr/cgi-bin/twiki/view/Picoforge/Web.

[32] *Seul*. http://www.seul.org.

[33] *OpenOffice*. http://www.openoffice.org.

[34] *Apache*. http://www.apache.org.

[35] *Kernel Archives*. http://www.kernel.org.

[36] *Mozilla Foundation*. http://www.mozilla.org.

[37] *K Desktop Environment*. http://www.kde.org.

[38] *Drupal*. http://www.drupal.org.

[39] *Gnome Foundation*. http://www.gnome.org.

[40] *Mono Project*. http://www.mono-project.com.

[41] *MediaWiki*. http://www.mediawiki.org.

[42] *Launchpad*. http://www.launchpad.net.

[43] B. Luthiger. Software-Entwicklung and der ETH Zurich, Internal report. August 2007.

[44] *Wikipedia Service Oriented Architectures*. http://en.wikipedia.org/wiki/Service-oriented_architecture.

[45] *Collanos*. http://www.collanos.com.

[46] *Eiffel*. http://www.eiffel.com.

[47] *Wikipedia Message passing*. http://en.wikipedia.org/wiki/Message_passing.

[48] J. Postel. Transmission Control Protocol. RFC 793. September 1981.

[49] *POSIX.1-90 System Application Program Interface (API) [C Language]. Information technologyÑPortable Operating System Interface (POSIX)*. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, 1990.

[50] *Goanna*. http://goanna.origo.ethz.ch.

[51] *Lighthttpd*. http://www.lighttpd.net.

[52] *fastCGI*. http://www.fastcgi.com.

[53] *EiffelStore*. http://www.eiffel.com.

[54] *MySQL*.

[55] *ODBC*. http://en.wikipedia.org/wiki/Open_Database_Connectivity.

[56] *Subversion source control system*. http://subversion.tigris.org.

[57] J. Postel. Simple Mail Transport Protocol. RFC 821. August 1982.

[58] *EiffelNet*. http://www.eiffel.com.

[59] *Manpage for KILL*.

[60] *Start-Stop-daemon man page*.

[61] *Valgrind*. http://www.valgrind.org.

[62] *Valgrind Converter*. http://eiffelroom.com/tool/valgrind_converter.

[63] *KCachegrind*. http://kcachegrind.sf.net.

[64] Dejan S. Milojicic, Vana Kalogeraki, and Rajan Lukose. *Peer-to-Peer Computing*. http://www.hpl.hp.com/personal/Dejan_Milojicic/p2p_o.pdf, July 2002. HP Laboratories, Palo Alto, HPL-2002-57.

[65] *JXTA JSE Platform*. http://platform.jxta.org.

[66] *JXTA v2.0 Protocols Specification*. http://spec.jxta.org/nonav/v1.0/docbook/JXTAProtocols.html.

[67] Bernard Traversat, Mohamed Abdelaziz, and Mike Duigou. *Project JXTA 2.0 Super-Peer Virtual Network*. http://www.jxta.org/project/www/docs/JXTA2.0protocols1.pdf, May 2003. Sun Microsystems, Inc.

[68] Brendon J. Wilson. *JXTA*. New Riders, Indianapolis, IN, USA, first edition, June 2002. http://www.brendonwilson.com/projects/jxta-book/.

[69] Daniel Brookshier, Darren Govoni, Navaneeth Krishnan, and Juan Carlos Soto. *JXTA: Java P2P Programming*. Sams, Indianapolis, IN, USA, first edition, March 2002.

[70] R. Moats. *URN Syntax*. RFC 2141 (Proposed Standard), May 1997.

[71] ISO (International Organization for Standardization). *9834-8:2004 Procedures for the operation of OSI Registration Authorities: Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 Object Identifier components*. ITU-T Recommendation X.667, September 2004.

[72] P. Leach, M. Mealling, and R. Salz. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122 (Proposed Standard), July 2005.

[73] N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. RFC 2046 (Draft Standard), November 1996. Updated by RFCs 2646, 3798.

[74] M. Murata, S. St. Laurent, and D. Kohn. *XML Media Types*. RFC 3023 (Proposed Standard), January 2001.

[75] Project JXTA. *SRDI: JXTA Shared Resource Distributed Index Design Plan*. http://platform.jxta.org/java/srdi.html, January 2006.

[76] *Gobo*. http://gobosoft.com.

[77] G. Blair and J.-B. Stefani. *Open Distributed Processing and Multimedia*. Addison-Wesley, December 1997.

[78] A. ShaikhAli, O.F. Rana, R. Al-Ali, and D.W. Walker. Uddie: An extended registry for web services. In *SAINT-W '03: Proceedings of the 2003 Symposium on Applications and the Internet Workshops (SAINT'03 Workshops)*, page 85, 2003.

[79] K. Pauls and T.G Bay. Reuse Frequency as Metric for Dependency Resolver Selection. In *Component Deployment: Third International Working Conference, CD 2005*, volume 3798, pages 164–176, 2005.

[80] P. Eugster and S. Baehni. Abstracting Remote Object Interaction in a Peer-to-Peer Environment. *Concurrency & Computation: Practice and Experience*, 17(7-8), June 2005.

[81] P. Eugster and R. Guerraoui. Distributed Programming with Typed Events. *IEEE Software*, 2(21):56–64, March 2004.

[82] M. Oriol and M. Hicks. Tagged Sets: A Secure and Transparent Coordination Medium. In *7th Int. Conf. on Coordination Models and Languages*, April 2005.

[83] C. Bryce, M. Oriol, and J. Vitek. A Coordination Model for Agents Based on Secure Spaces. In *3rd Int. Conf. on Coordination Models and Languages*, pages 4–20, April 1999.

[84] P. Gregono and M. Sakkinen. Copying and Comparing: Problems and Solutions. In *14th European Conference on Object-Oriented Programming (ECOOP 2000)*, pages 226–250, June 2000.

[85] T.G. Bay, P. Eugster, and M. Oriol. A First Order Model of Component Lookup. Technical report, Swiss Federal Institute of Technology in Zurich (ETHZ), 2006.

[86] P. Eugster and R. Guerraoui. Probabilistic Multicast. In *3rd IEEE International Conference on Dependable Systems and Networks (DSN 2002)*, pages 313–323, June 2002.

[87] *PEAR Package XML_RPC*. http://pear.php.net/package/XML_RPC.

[88] *GeSHi - Generic Syntax Highlighter*. http://qbnz.com/highlighter.

[89] *Google Analytics*. http://www.google.com/analytics.

[90] *Google Custom Search Engine*. http://www.google.com/coop/cse.

[91] *PEAR Package Text_Wiki*. http://pear.php.net/package/Text_Wiki.

[92] *PEAR Package Text_Diff*. http://pear.php.net/package/Text_Diff.

[93] Bertrand Meyer. The outside-in method of teaching introductory programming. In Manfred Broy and Alexandre V. Zamulin, editors, *Ershov Memorial Conference*, volume 2890 of *Lecture Notes in Computer Science*, pages 66–78. Springer, 2003.

[94] Keith J. O'Hara and Jennifer S. Kay. Open source software and computer science education. *J. Comput. Small Coll.*, 18(3):1–7, 2003.

[95] Marty J. Wolf, Kevin Bowyer, Don Gotterbarn, and Keith Miller. Open source software: intellectual challenges to the status quo. In *SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 317–318, New York, NY, USA, 2002. ACM Press.

[96] Eric Allen, Robert Cartwright, and Charles Reis. Production programming in the classroom. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 89–93, New York, NY, USA, 2003. ACM Press.

[97] Christopher P. Fuhrman. Appreciation of software design concerns via open-source tools and projects. In *10th Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts, at 20th European Conference on Object Oriented Programming (ECOOP)*, Nantes, FR, July 2006.

[98] David Carrington and Soon-Kyeong Kim. Teaching software design with open source software. *Frontiers in Education*, 3(33):S1C– 9–14, November 2003.

[99] Falko Rheinberg, Regina Vollmeyer, and Bruce D. Burns. QCM: A questionnaire to assess current motivation in learning situations. *Diagnostica*, 47:57–66, 2001.

[100] Paul R. Pintrich. A motivational science perspective on the role of student motivation in learning and teaching contexts. *Journal of Educational Psychology*, 95(4):667–686, 2003.

[101] *EiffelMedia*. http://eiffelmedia.origo.ethz.ch.

[102] *Heise article on Origo*. http://www.heise.de/newsticker/meldung/94910.

[103] *Nagios*. http://www.nagios.org.

[104] *Mirmon*. http://people.cs.uu.nl/henkp/mirmon.