TECHNISCHE
UNIVERSITÄT
DARMSTADT

A FULLY DECENTRALIZED, PEER-TO-PEER BASED VERSION CONTROL SYSTEM

Vom Fachbereich Elektrotechnik und Informationstechnik
der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)
genemigte

DISSERTATIONSSCHRIFT

von

DIPL.-INFORM. PATRICK MUKHERJEE

geboren am 1. Juli 1976 in Berlin

Erstreferent: Prof. Dr. rer. nat. Andy Schürr
Korreferent: Prof. Dr.-Ing. Bernd Freisleben

Tag der Einreichung: 27.09.2010
Tag der Disputation: 03.12.2010

Hochschulkennziffer D17
Darmstadt 2011

Посвећено мојој Сандри,
која ме увек и у сваком погледу подржава и чини мој живот испуњеним.

ABSTRACT

Version control is essential in collaborative software development today. Its main functions are to track the evolutionary changes made to a project's files, manage work being concurrently undertaken on those files and enable a comfortable and complete exchange of a project's data among its developers. The most commonly used version control systems are client-server based, meaning they rely on a centralized machine to store and manage all of the content. The obvious drawbacks involved by bringing an intermediary server into the system include having a *single point of failure*, a *central ownership and vulnerability*, *scalability issues*, and *increased synchronization and communication delays*. Many popular global software projects switched to the emerging distributed version control systems, demonstrating the urgent need for a more suitable version control system.

This thesis proposes a fully decentralized, peer-to-peer based version control system as a solution to overcome issues of currently available systems. The peer-to-peer communication paradigm proved to be successful in a variety of applications. A peer-to-peer system consists of autonomous participants, thus certain behavior cannot be guaranteed. Its unreliable nature, however, means its usage in version control systems is not obvious. In order to develop a solution based on the peer-to-peer communication paradigm, existing client-server, distributed, and peer-to-peer version control systems are analyzed and evaluated using a set of requirements, which were derived from two realistic usage scenarios. Furthermore, the design details of those systems are closely examined in order to realize the impact of their design decisions on both functional features and quality aspects of the system, with the strongest focus on consistency. The proposed system is designed and implemented based on these findings.

The resulting system, PlatinVC, is a fully decentralized system, which features the complete system view of centralized systems, while overcoming their drawbacks. All workflows of existing version control systems, centralized or distributed, are supported. PlatinVC even supports a hybrid setup, where other version control systems can interoperate, using PlatinVC as a mediator. Moreover, it introduces an *automatic isolation* of concurrent work, which separates relevant and required updates from possibly unneeded ones. In this way, branches are only created when necessary. The evolutionary changes of links between files, which can be enhanced by any attributes, are also tracked. That extends snapshot-based version control to other purposes, e.g. traceability of software artifacts. PlatinVC is a serious alternative to current version control systems, as industry proven components for the most critical parts of the system were reused. The evaluation shows that PlatinVC meets the identified requirements completely, thereby being the first fully decentralized version control system that provides a high consistency degree, efficiency, and robustness.

The impact of this thesis is twofold: First, it offers the novel concept of automatic concurrent work isolation, so that the essential updates are separated from the unnecessary ones and the costs of integrating and merging branches are minimized. Second, this thesis provides the proof that the peer-to-peer paradigm can be pushed beyond its currently reputed limits by adding features, which previously seemed incompatible.

ZUSAMMENFASSUNG

Die Versionsverwaltung ist in der kooperativen Softwareentwicklung heutzutage unerlässlich. Ihre wichtigsten Funktionen sind die Nachverfolgung von evolutionären Änderungen an Dateien eines Projektes, die Koordination von nebenläufiger Arbeit an diesen und es den Entwicklern zu ermöglichen, die kompletten Daten eines Projektes auf komfortable Weise auszutauschen. Die am häufigsten verwendeten Versionsverwaltungssysteme sind Client-Server-basiert, was bedeutet, dass sie einen zentralen Rechner zur Speicherung und Verwaltung aller Daten voraussetzen. Die offensichtlichen Nachteile, die durch das Einbringen eines zwischengeschalteten Servers in der Kommunikation der Entwickler verursacht wird, sind die Schaffung einer *zentralen Ausfallstelle (single point of failure)*, einer *zentralen Angriffsstelle* mit *zentralen Eigentumsverhältnissen (central ownership and vulnerability)*, *Skalierbarkeitsproblemen* und *erhöhte Synchronisations- und Kommunikationszeiten*. Viele populäre globale Software-Projekte sind deshalb zu den aufkommenden verteilten Versionsverwaltungssystemen gewechselt, was die dringende Notwendigkeit für ein passenderes Versionsverwaltungssystem verdeutlicht.

Diese Arbeit stellt ein völlig dezentrales, Peer-to-Peer basiertes Versionsverwaltungssystem vor, dass die Probleme aktueller Systeme überwindet. Das Peer-to-Peer Kommunikationsparadigma ist in einer Vielzahl von Anwendungen sehr erfolgreich. Dieses Paradigma beschreibt ein System, das ausschließlich aus autonomen Teilnehmern aufgebaut ist. Dadurch kann kaum bestimmtes Verhalten garantiert werden, was den Einsatz von Peer-to-Peer Kommunikation für ein Versionsverwaltungssystem äußerst erschwert. Um eine Lösung zu entwickeln wurden bestehende Client-Server, verteilte und Peer-to-Peer basierte Versionsverwaltungssysteme bezüglich einer Menge von Anforderungen analysiert und ausgewertet, die von zwei realistischen Anwendungsszenarien abgeleitet wurden. Darüber hinaus sind die Design-Details dieser Systeme genauestens untersucht worden, um die Auswirkungen der getroffenen Entscheidungen auf Funktionalität und Qualitätsaspekte des Systems zu erkennen, insbesondere die Auswirkung auf die vom System gebotene Konsistenz. Basierend auf diesen Erkenntnissen wurde das vorgeschlagene System konzipiert und umgesetzt.

Das resultierende System, PlatinVC, ist ein völlig dezentrales Systems, dass die ganzheitliche Systemsicht zentraler Systeme anbietet, ohne deren Nachteile zu erben. Alle in bisherigen Versionsverwaltungssystemen üblichen Arbeitsabläufe werden unterstützt, unabhängig davon, ob diese bei zentralen oder dezentralen Systemen verwendet werden. PlatinVC kann es sogar in einem hybriden Aufbau als Vermittler anderen Systemen ermöglichen, sich interoperabel auszutauschen. Darüber hinaus führt diese Arbeit ein Konzept zur *automatischen Isolierung* von nebenläufigen Änderungen ein, bei dem benötigte Beiträge von potenziell unnötigen getrennt werden. Zweige werden nur angelegt, falls sie tatsächlich notwendig sind. Die evolutionären Änderungen von Links zwischen Dateien, die mit beliebigen Attributen angereichert werden können, werden ebenfalls verwaltet. Dies erweitert die Nutzung über snapshot-basierter Versionsverwaltung hinaus für andere Zwecke, wie beispielsweise die Nachverfolgung von Software Artefakten (traceability). PlatinVC ist eine ernstzunehmende Alternative zu aktuellen Versionsverwaltungssystemen, da für die kritischsten Systemteile Komponenten wiederverwendet wurden, die sich im industriellem Umfeld bewährt haben. Die Evaluationsergebnisse zeigen, dass PlatinVC die identifizierten Anforderungen komplett erfüllt und somit das erste völlig dezentrale Versionsverwaltungssystem ist, das eine hohe Konsistenz, Effizienz und Robustheit bietet.

Der Einfluss dieser Arbeit zeigt sich in zwei Formen: Erstens bietet sie ein neuartiges Konzept zur automatischen Isolation nebenläufiger Arbeit, so dass notwendige und nutzlose Updates separiert und die Kosten der Integration und Zusammenführung von Verzweigungen minimiert werden. Zweitens beweist diese Dissertation durch bisher für inkompatibel gehaltene Funktionen, dass das Peer-to-Peer Paradigma über seine momentane Anwendungszwecke hinaus eingesetzt werden kann.

"Sometimes when you innovate, you make mistakes.
It is best to admit them quickly, and get on with improving your other innovations."

— Steve Jobs —

PREFACE

Rationale

During my work as a consultant for an international corporation, I experienced the magnitude and importance of collaboration and team work. Multiple code and text files are subject to concurrent changes by multiple developers. Tracking the evolution of a project and managing possible conflicts are essential features of a development environment. I used version control systems daily to recover from mistakes and revert to earlier version of the files, to identify potentially incompatible changes, resolve those conflicts, and use it as part of a backup strategy. Even while developing my own version control system with several students of mine in the last 4 years of my research, I relied heavily on a version control system. In the development of large software systems, such as operating systems, thousands of developers work on one development project, distributed not only across one building or one city but across the globe.

The more I have worked with version control systems, the more I have seen how unsuitable they are to the distributed and large-scale nature of developers. Truth be told, we, computer scientists, will always find many limitations of the software we use and have plenty of improvement ideas. There were, however, too many questions becoming apparent while working with current version control systems. Why is it that a repository is unavailable when needed the most, which is something that should never occur to the crucial point of communication for developers? Why does the reliability of a repository depends entirely on a single point, a centralized server? Why must I wait so long till my contributions are applied to the repository? I had the great opportunity to do my doctorate in the field of peer-to-peer networking and software engineering, the combination of which is key to solve those limitations. My research was partially founded by the DFG Research Group QuaP2P (FOR 733) with the focus being on improving the quality of peer-to-peer systems: This was the driving force behind my additional mission to push the peer-to-peer paradigm to its limits. In this thesis I answer the questions I posed above, analyze the state of the art version control systems and propose a suitable solution.

Conventions Used in This Thesis

In this thesis, whenever a term needs further explanation, it is marked in ***italics-bold*** printed letters and defined in a ***definition box*** on its first appearance.

Definition Box

In such a **definition box**, a chosen term is explained further. For each box there exists an entry in the index, which can be found in the appendix of this work.

The reader is advised to look up an ambiguous term in the index of this work, which is presented in the last pages. A number in the index refers to the page where the corresponding definition box can be found.

In spite of the fact that this thesis is single-authored, I felt it was more appropriate to write it in the first-person plural. This thesis would not exist without the help and support from the people to whom the acknowledgements are due.

DANKSAGUNG

Mit den letzten Worten, die ich in dieser Dissertation niederschreibe, danke ich allen, die mich auf dem Weg der Promotion auf die eine oder andere Weise unterstützt haben.

Ungewöhnlich früh in dieser Danksagung danke ich Sandra, die mich auch ungewöhnlich umfassend unterstützt hat.

Der größte Dank gilt meinem Doktorvater Andy Schürr. Ohne Deine Betreuung von der ersten bis zur letzten Stunde wäre die Qualität meiner Forschung sicherlich nicht so hoch. Immer stand Deine Tür für mich offen, wo aus einem "hast Du mal 5 Minuten Zeit?" oft eine Stunde wurde. Ich danke auch Bernd Freisleben. Mit Ihrer herzlichen Art habe ich mich schon beim ersten Treffen wohl gefühlt und die beruhigenden Ratschläge haben meine Nerven am Tag der Disputation geschont. Dir, Ralf Steinmetz, danke ich für die warme Aufnahme bei KOM - sowohl damals als Projektpartner, als auch später. Vielen Dank, dass ich den Feinschliff an meiner Arbeit bei Dir fertigstellen durfte.

Dank gebührt auch meinen Kollegen bei ES. Für eine angenehme Atmosphäre, guten Tipps und beistehenden Worten Danke ich meinen Zimmerkollegen Tobias Rötschke und Sebastian Oster. Auch danke ich Karsten Saller, meinem (thematischen) Nachfolger, der immer zu helfen bereit ist. Besonders Felix Klar und Ingo Weisemöller danke ich für die vielen Male, die Ihr mir bei Metamodellierungsproblemen mit Antworten geholfen habt.

Oliver Heckmann, Nicolas Liebau und Kálmán Graffi - die alle von Kollegen zu Freunde geworden sind: Nico hat mir erst die Idee zum Promovieren in Darmstadt in den Kopf gesetzt, und dabei mit Tipps und freizeitlicher Entspannung geholfen. Oliver hat immer guten Rat gehabt - es hat mich besonders in den ersten Jahren gefreut, dass Du auch mein Büro besucht hast, um mir helfende Ratschläge zu geben. Zusammenarbeit mit Kálmán war im Projekt eigentlich nicht geplant - dennoch war es die ergiebigste und ausführlichste. Christian Gross und Max Lehn danke ich in erster Linie für die erste Implementierung meiner Forschungsidee - damals noch als Wiki getarnt. Sebastian Kaune, Konstantin Pussep, Dominik Stingl und Osama Abboud danke ich für die herzliche Aufnahme als Peer. Meinen QuaP2Plern danke ich für die kritischen Diskussionen - allen voran Christof Leng, der mir für einiges die Augen geöffnet hat. Thorsten Strufe kam gerade rechtzeitig nach Darmstadt, um mich für die finale Phase meiner Promotion aufzubauen, durch guten Rat und entspannenden Unternehmungen.

All meinen Studenten möchte ich für die fruchtvolle Zusammenarbeit danken. Ganz besonders Sebastian Schlecht. Neben großem Dank für die Hilfe bei der Implementierung möchte ich Dir für Dein großes Engagement Danken, das über allen Erwartungen weit hinausgeht.

Vielen Dank auch bei den unterstützenden Kollegen bei KOM. Besonders möchte ich folgenden Personen danken: Karola, für die Hilfe beim Organisieren - von Projektbegehungen zu meiner Promotionsfeier. André Miede, nicht nur für die wunderschöne Dokumentvorlage, auch für die nicht müde werdende Unterstützung beim Anpassen selbiger. Warm thanks go to our friend Annabel Baird, who proof read my thesis. You did a magnificent job!

Zu guter Letzt möchte ich meiner Familie danken. Meinem Vater, Pradip Mukherjee, dessen Worte mich erst so weit gebracht haben. Meiner Mutter Margret Mukherjee, die mir immer bedingungslos für mich da ist. Meinen Schwestern Anita und Sonali, die trotz der räumlichen Distanz die Familienbande zusammenhalten, Christian für die Unterstützung bei meinen Pflichten als Sohn und meinen Nichten Helena Pari und Johanna Shalin für die herzhaften Momente. Salomon für die Gelegenheit mich im Gespräch über meine Hobbies zu verlieren. Sale, Zaga und Zlatko danke ich für die aktive Unterstützung bei Euren Besuchen in Darmstadt. In Eurem Urlaub habe ich mich wie im Urlaub gefühlt.

Dieser Abschnitt erforderte ähnliche mentale Kraft wie die anderen Teile dieser Dissertationsschrift: Wie leicht vergisst man doch Personen, den man überaus dankbar ist! Wenn Du, lieber Lesen, Dich angesprochen fühlst, dann sei Dir gewiss, dass ich Dir nur versehentlich nicht in diesem Text gedankt habe.

CONTENTS

I INTRODUCTION 1

1 Introduction 3
- 1.1 Motivation 4
- 1.2 Vision 5
- 1.3 Challenges 6
- 1.4 Goal 7
- 1.5 Outline 7

2 Suitability of Peer-to-Peer Paradigm for Application Scenarios 9
- 2.1 Peer-to-Peer Communication Paradigm 9
- 2.2 Wiki Engines 11
  - 2.2.1 Cooperation in a Wiki Scenario 12
- 2.3 Global Software Development 14
- 2.4 Benefits of Peer-to-Peer Paradigm for the Application Scenarios 16
  - 2.4.1 Shortcomings of Server-centric Solutions 17
  - 2.4.2 Characteristics of Peer-to-Peer based Solutions 18
- 2.5 Running Example 20
- 2.6 Summary 21

3 Requirements and Assumptions 23
- 3.1 Assumptions 23
- 3.2 Requirements 24
  - 3.2.1 Functional 24
  - 3.2.2 Non-functional 27
  - 3.2.3 Security Aspects 29
- 3.3 Summary 30

II VERSION CONTROL SYSTEMS 33

4 Foundations of Version Control Systems 35
- 4.1 Collaboration Workflows 35
- 4.2 The Frequency of Committing Changes 37
- 4.3 Configuration Management 38
- 4.4 Consistency and Coherency in Version Control Systems 38
  - 4.4.1 Terminology 39
  - 4.4.2 Degrees of Consistency 40
  - 4.4.3 Coherency in Version Control Systems 42
- 4.5 Summary 43

5 Notable Version Control Systems 45
- 5.1 Centralized Version Control 45
  - 5.1.1 SCCS 45
  - 5.1.2 RCS 46
  - 5.1.3 CVS 46
  - 5.1.4 SVN 48
  - 5.1.5 ClearCase 49
- 5.2 Distributed Version Control 52
  - 5.2.1 Basic Architecture of dVCS 52
  - 5.2.2 Monotone 55
  - 5.2.3 Git 55
  - 5.2.4 Mercurial 58
- 5.3 Peer-to-Peer Version Control 60
  - 5.3.1 Wooki 61

5.3.2 DistriWiki 62
5.3.3 CVS over DHT 63
5.3.4 Code Co-op 64
5.3.5 Pastwatch 65
5.3.6 GRAM 68
5.3.7 SVCS 69
5.3.8 Chord based VCS 69
5.4 Systems for Collaborative Work Support 70
5.5 Summary 71
6 Analysis of Related Version Control Systems 73
6.1 Realized Requirements per System 73
6.2 Analysis of System Properties Leading to Fulfilled Requirements 75
6.2.1 Functional Requirements 75
6.2.2 Non-functional Requirements 76
6.2.3 Security Aspects 80
6.3 Degree of Consistency 80
6.4 Taxonomy of Key Mechanisms 82
6.4.1 Concurrency Control 82
6.4.2 Repository Distribution 82
6.4.3 Repository Partitioning 83
6.4.4 Communication Paradigm 84
6.4.5 Communication Protocol 85
6.5 Promising Mechanisms 85
6.5.1 Concurrency Control 85
6.5.2 Repository Distribution 87
6.5.3 Repository Partitioning 87
6.5.4 Communication Paradigm 88
6.5.5 Communication Protocol 88
6.6 Summary 88

III PEER-TO-PEER VERSION CONTROL SYSTEM - PLATINVC 91
7 Overview of PlatinVC 93
7.1 Basic Architecture 93
7.2 Workflow 94
7.2.1 Frequency of Commits 94
7.2.2 Repository Sharing 94
7.2.3 Conflict Resolution 95
7.3 Features 97
7.3.1 Automatic isolation of concurrent work 97
7.3.2 Working Offline 100
7.3.3 Interoperability 100
7.3.4 Offers all dVCS Operations 100
7.3.5 Backing up Artifacts Redundant 101
7.3.6 Degree of Consistency 101
7.3.7 Support for Traceability Links 102
7.4 Services 103
7.4.1 Modul Management Operations 103
7.4.2 Retrieve Operations 104
7.4.3 Share Operations 106
7.5 Summary 107
8 Design of PlatinVC 109
8.1 Design Principles 109
8.2 Design Overview 110
8.3 Choice of the Supporting Systems 111

8.3.1 Component Intercommunication & Lifecycle Management 111
8.3.2 Communication network layer 111
8.3.3 Local version control mechanisms 112
8.4 Global Version Control Mechanisms 113
8.4.1 Storage Components on each Peer 113
8.4.2 Repository distribution and partitioning 115
8.4.3 Replication 117
8.4.4 Collaboration 118
8.4.5 Retrieve Updates 122
8.4.6 Share Versions 130
8.4.7 Conflict Handling 137
8.4.8 Additional mandatory mechanisms 137
8.5 Maintenance Mechanisms 138
8.6 Failure Handling 138
8.6.1 Handling Network Partitioning 139
8.6.2 Handling Indeterministic Routing 140
8.6.3 Recovery of Missing Snapshots 141
8.7 Traceability Links 141
8.8 Summary 141
9 Prototypical Software Development Environment 143
9.1 Modular Development of Peer-to-Peer Systems 143
9.2 Framework Services 144
9.2.1 Communication Management 144
9.2.2 Security 145
9.2.3 Storage 148
9.3 User Level Applications 148
9.3.1 Eclipse IDE 148
9.3.2 PlatinVC- a Peer-to-Peer based version control system 149
9.3.3 Piki - a Peer-to-Peer based Wiki Engine 150
9.3.4 Askme- Peer-to-Peer based Aware Communication 151
9.4 Summary 152
10 Evaluation 155
10.1 Evaluation Goals 155
10.1.1 Quality Aspects 155
10.1.2 Metrics 156
10.2 Evaluation Methodology 157
10.2.1 Evaluation Environment 158
10.2.2 Evaluation Platform 158
10.3 Workload 159
10.3.1 Number of Users 159
10.3.2 Experiment Data 159
10.3.3 User Behavior 160
10.3.4 Churn Model 160
10.3.5 Experiment Timeline 160
10.3.6 System Parameter Settings 161
10.4 Evaluation Results 161
10.4.1 Consistency Degree 162
10.4.2 Robustness 162
10.4.3 Freshness 163
10.4.4 Duration of Push and Pull 164
10.4.5 System Load 166
10.5 Comparative Evaluation 166
10.6 Summary 167

IV FINALE 171
11 Conclusion 173
11.1 Summary and Conclusions 173
11.2 Contributions 175
11.3 Outlook 176
11.4 Implications 177

BIBLIOGRAPHY 179
CURRICULUM VITÆ 189
PUBLICATIONS 193

V APPENDIX 195
A About the Prototype's Development 197

Erklärung 199
List of Figures 201
List of Tables 202
Index 203

Part I

INTRODUCTION

Development, relating in particular to project work, evolved from single site work into a globally spread fashion, in which multiple persons collaborate from physically distant places. The utilized version control systems were created in times when a project's development was concentrated on one physical location.
A motivation with a vision for a better suiting solution to support this evolved form of project work is laid out in Chapter 1 along with an overview of our goals and the challenges that arose. We take a deeper look at two scenarios where the distributed nature of the development suffers from the current limitations of the available version control tools in Chapter 2. A complete set of the requirements that a beneficial solution must fulfill are elaborated in Chapter 3.

1 INTRODUCTION

The Internet has become the main means of communication in today's society. It has a huge impact on people across the world. People send e-mails, chat, talk, exchange various information, multimedia data, all over Internet. We are also witnessing new trends of communication, using popular online social networks (e.g., Facebook [fac]), the sharing of personal content - expertise, experiences, thoughts (e.g., Blogs [Bloo4]), articles (e.g., Wikipedia [wikb]), and videos (YouTube [You]). The vast majority of people now consult wikipedia articles to briefly inform themselves on subjects of interest or watch tutorials on various topics from YouTube users. The sharing of various types of data, communication, and collaboration are the common features of all popular and now essential Internet applications.

Means of efficient and powerful communication and collaboration do not only have a social impact, but are also crucial for industry. Let us examine, for instance, multinational corporations. They are spread across the globe and distribute their work over many distant locations. Efficient collaboration tools are not only essential but crucial for running such corporations. For instance, Mercedes Benz design their cars in Germany but hold assembly lines in many countries around the world. A lack of appropriate collaboration platforms could lead to huge costs and even endanger drivers' lives, if misunderstandings arise. Product development in software companies is also globally distributed. For instance, Google, whose headquarters are in MountainView, California, has offices in 37 countries in Europe, Latin America, Asia, Australia, and Middle East [goob]. One reason for this distribution is to assemble expertise from other countries, (e.g., Google research center Zürich in Switzerland), however, reducing costs (e.g., development center in Bangalore, India) is certainly of interest. Many documents can be shared online in an efficient way, with no communication overhead (e.g., e-mails, phone calls). A designer of a new Mercedes Benz E-Class model needs only to upload his specifications to the collaboration platform so that the worker in the assembly line on the other side of the globe can always have access to the latest version of design documents. Collaborators can be distributed all around the world and in addition to issues involved with working in different time zones, the Internet communication delays can be huge if the communication in the provided platform is inefficient. Any ***outdated*** data can lead to huge risks to the success of a project.

Particularly in the software industry, which will remain the focus of this thesis, developers collaborate intensively and often work on the same project or file at the same time. It is, therefore, very important to enable control over possible conflicts. Mistakes in development can be eliminated by going back to earlier stages of the process. Tracking evolutionary changes and managing concurrent work on the same document is crucial for such a collaboration.

Version Control System

A **version control system** tracks evolutionary changes to files. The so called **commit** operation detects and records changes a user has made to her files in her ***working copy***. These changes can be retrieved by any other developer at any time later with the **update** operation. The changes are listed in chronological order with additional information, such as the author's name and comments.

Version control systems also manage concurrent work. When multiple authors modify a file at the same time, conflicts can happen. Using a version control system those conflicts are detected and resolved.

In summary, there are three crucial requirements of version control system:

1. The possibility to revert the state of an entire project or a single file to any point of earlier development stage, an earlier version. This feature provides key support for correcting mistakes.

2. Coordinate the work of an unlimited number of people on the same project or same file without the overhead of constant manual back and forth file exchange.

3. Managing possible conflicts resulting from the concurrent work of many developers on the same project or file at the same time.

The question raised at this point is: What kind of collaboration platforms exist today and do they fulfill the aforementioned needs?

1.1 MOTIVATION

The sharing of documents and files is now supported by various tools. Most of them, such as the popular platforms dropbox [dro] and Google docs [gooa], support only the second of the aforementioned benefits. The Dropbox client, for example, allows users to drop any file into a designated folder, which is then synchronized directly to any of the user's other computers with the Dropbox client. Google docs supports collaborative text editing.

Common to all of them, however, is that they all suffer from a complete lack of or serious limitations to full version control - tracking only the evolutionary changes of single files.

The most widely used version control systems, such as CVS [CVS] and SVN [Pil04], provide support for all three requirements. They belong to the vital tools in multi-developer software projects. Their usage not only eases collaboration but also speeds up the development, through the possibility of correcting any mistake by restoring the project to any earlier stage. Looking deeper into the technology behind them, we can see that those version control systems use a centralized point of communication and storage. It is a server that is always online and stores all data, responds to all user's requests and it plays an intermediary role between users. The basic communication flow is presented in Figure 1. A developer uploads data on a server which tracks the changes, marks the versions, and manages the conflicts. All other developers receive the changes from the server.

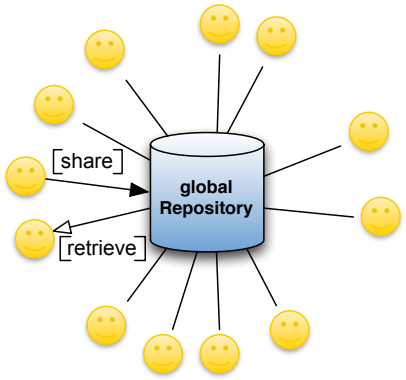


Figure 1: Basic principle of centralized version control systems

An obvious drawback to this communication principle is that a server presents a *single point of failure*. In the case of a server break down due to, e.g., overload, power outage, or an attack, the complete collaboration platform fails and developers are hindered in their work on a project. Additionally, it also represents a *single point of control and ownership*. If such a version control system is used by a huge number of developers at the same time, it demonstrates evident *scalability issues* as a server may run out of resources leading to its failure or, in best case, poor performance. As already mentioned, developers can reside all around the world. The centralized communication model, also called **client-server**, can lead to unequal Internet delays in such a globally distributed environment, bringing additional *synchronization and communication delays*. Last but not least, having a centralized server as an essential system component, brings additional maintenance costs for the dedicated hardware.

We explained the need for version control systems in collaborative work nowadays and understood the limitations of currently used tools. Next, we will present the vision of this thesis which shows how we can overcome these limitations and provide even better support for collaborative software development.

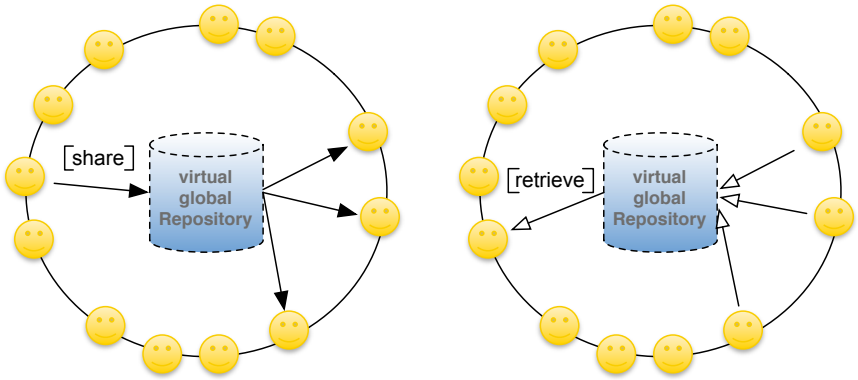
1.2 VISION

If we look at the inherent distribution of developers, we first notice that centralized communication in existing version control systems is orthogonal to it. A more natural way of communication would be decentralized, directly among the users. Such a communication paradigm is called ***peer-to-peer***.

Peer-to-Peer (p2p)

The term **peer-to-peer** describes the "equal-to-equal" property of this communication model. Opposite to the centralized, client-server communication model, all users, so called **peers**, are both consumers and contributors to the system, acting as both servers and clients at the same time. They share their resources (e.g., data, memory, CPU power) directly, without an intermediary server. The users are, therefore, called peers, as they have the same rights and responsibilities (which is not a strict rule but more of an initial, basic principle).

Peer-to-peer became a popular means of communication, storage, and sharing in the Internet after its huge success in 1999, with the notorious file-sharing application Napster [Shio1]. Since then, this fully decentralized communication paradigm has proven its benefits in many applications beyond file-sharing, such as Voice-over-IP (e.g., Skype [Skyb]), video streaming (e.g., SopCast [Sop]), and content distribution (e.g., BitTorrent [Coho3]). Some of the main advantages of using peer-to-peer communication paradigm are very *low maintenance costs*, *no single point of failure*, *scalability*, and *collective ownership*.



(a) share changes (b) get updates

Figure 2: Basic principle of peer-to-peer version control systems

Combining the peer-to-peer paradigm with a version control system solves many issues current version control systems suffer from. In addition to that, this thesis introduces *a novel collaboration workflow*, which solves another issue in current version control systems, not previously mentioned. Let us observe the situation when several developer teams work on the same project. Each of them focuses on the implementation of different system features. To avoid confusions and inevitable programming mistakes caused by one team working on one feature, obstructing the development of another team working on another feature, for each team a so called ***branch*** is created. With branches commits and updates of all developer teams are separated. When teams accomplish their tasks, the resulting code must be ***merged*** into a functional project.

Merging

Whenever the same ***artifact*** is developed concurrently the changes have to be merged. If the modifications happened in different lines they can be combined automatically. Nevertheless a user should check the result of the automatic merge process as the resulting version might have semantic errors. There are more sophisticated merge approaches, which can take the semantic of specific types of artifacts (e.g., Java source code) into account. However, those merge algorithms were never used widespread. A special form of a commonly used merge algorithm is the **three way merge**, in which the shared base version of the conflicting artifact versions is consulted as well. In this algorithm actual changes can be identified, which overwrite unchanged lines.

This is not a simple task. For instance, before the branches are merged, many helpful fixes in common files of all branches are not shared. Therefore, each team may solve the same bugs in different ways, losing their time and making the integrations of those changes troublesome. Merging branches is problematic - the longer it is postponed, the more conflicts might arise. Optimally these conflicts could be resolved by the respective authors in the most efficient and valid way. Different implementations resulting in the same effect are very challenging to identify and merge.

Our collaboration workflow introduces ***automatic isolation of concurrent work***, a mechanism which avoids the aforementioned problem by automatically separating the work of teams while notifying about potentially relevant updates. Two teams that, for instance, work on the same file, will receive the updates from each other although they work on separate tasks. When a developer is interested in the latest versions of a set of files, she specifies a folder that should be kept updated. Updates are *snapshot-based*, meaning that if they are integrated, all related changes are provided as well. In that way, she can integrate updates from other teams into her versions immediately or at a later date. In the latter case, an *isolation* (analog to a branch) is automatically created so that every developer profits from the decision to separate ***development lines***. It is no longer necessary to create (often unnecessary) branches in advance, which might cause problematic change integration.

The basic communication flow in our peer-to-peer version control system is visualized in Figure 2. For the first step, developers stores their changes locally, on their computer. As soon as they consider the state of their code to be stable and ready for sharing, they will *push* the changes (see Figure 2a) to the specific group of peers assigned to manage the changed file. Developers can update their local files to the latest version by *pulling* them from the peer-to-peer network, which for them appears as a central, global repository instance (marked using dotted lines on "global repository" on the figures). Without being aware of it, they actually send requests to the peers assigned for storing all changes of the corresponding file. A users can decide to receive only the updates to folders they specify, which in contrast to client-server version control systems retrieves updates on related files as well. Automatic isolation is initiated when developers decide to separate conflicting development lines. The natural separation of project parts into subfolders can be utilized in work isolation, which eliminates the need for explicit branches.

There are, however, many challenges which make the use of peer-to-peer in version control challenging.

1.3 CHALLENGES

In spite of the fact that the peer-to-peer paradigm continuously proves its success in a plethora of applications, it can have some limitations. They are caused by the main characteristics of this communication paradigm:

- Participants are fully *autonomous*, meaning they join and leave the system at any time.
- There is *lack of centralized control and view* on the system.

The core of a peer-to-peer system is the peers themselves, the availability of the offered services if fully dependent on them. In the sudden absence of many peers, a system can guarantee neither availability of its services nor its quality. All peers are equal in terms of their rights and responsibilities within in the system. That means that no peer has a role of higher importance than another, as is the case for a server in client-server solutions. Finding the right peers with which to share updates without having a global view of the system is a challenging task.

Two main challenges were faced when bringing the peer-to-peer paradigm to a version control system:

- *A Lack of global knowledge*, which is evidently crucial when pushing and pulling updates and finding the appropriate group of peers for particular updates. Finding a particular peer in fully decentralized fashion, with no centralized view of the system is a challenging task in itself.
- The *Reliability* of the system and its performance due to the unpredictable peer behavior. Is it possible to create a reliable system dependent on unreliable, autonomous peers?

This thesis overcomes these challenges and proves that the peer-to-peer paradigm can be pushed beyond its current boundaries, contributing its inherent strong points and adding seemingly unfeasible features.

From the motivational problem of current version control systems, the vision of the solution, and the rising research challenges, we define the main goal of this thesis.

1.4 GOAL

The main goal of this thesis is to provide a *fully decentralized, peer-to-peer version control system that provides suitable / full support for distributed software development*.

In order to achieve this goal, the following objectives are addressed in this thesis:

- *Requirements analysis* by examination of feasibility, benefits and drawbacks of the peer-to-peer communication paradigm in our two application scenarios, namely wiki and global software development. The aims are to identify "the best of both worlds", client-server and peer-to-peer, by retaining the advantages of both and eliminating their drawbacks.
- *Evaluation of existing solutions* where the most important version control systems are investigated and compared, to learn from their design decisions and the impacts on the resulting features and quality.
- *Design of fully decentralized peer-to-peer version control system* with all mechanisms, protocols, and algorithms needed to fulfill the stated requirements, being both functional and quality requirements.
- *Evaluation* of the solution by providing a running and tested proof-of-concept and performing testbed experiments to address various quality aspects.
- Proving feasibility of *pushing the peer-to-peer paradigm beyond its current boundaries*. Appropriate mechanisms will be proposed to overcome currently unsolved issues in peer-to-peer applications, which present a great obstacle to an even broader usage of the peer-to-peer technology.

1.5 OUTLINE

This work consists of four main parts. They are structured as following:

Part i first introduces the motivation for the subject of version control and discusses the problem set out in this chapter. In the following chapterwe discuss two application scenarios

that can benefit from the peer-to-peer paradigm and derive functional and non-functional requirements in Chapter 3.

When the requirements are defined, we focus on version control systems in Part ii. First the foundations of version control are explained in Chapter 4. Existing solutions for version control are discussed in detail in Chapter 5 and they are analyzed according to the requirements, design decisions and their impact in Chapter 6.

Part iii is the crucial part of this thesis as it presents the proposed peer-to-peer version control system. Chapter 7 describes the basic architecture and workflows in the system. A detailed look into the design of the system through comparative representation of chosen and alternative design decisions is given in Chapter 8. A proof of concept and its implementation details are explained in Chapter 9 and experiments running on a testbed to evaluate the required quality aspects are elaborated in Chapter 10.

The concluding Part iv summarizes the key contributions of this dissertation, implications for collaborative software development and new research challenges arising from this work.

2 SUITABILITY OF PEER-TO-PEER PARADIGM FOR APPLICATION SCENARIOS

This chapter discusses two application scenarios and explores the suitability of the peer-to-peer paradigm for both of them. *Wiki* and *global software development* are observed as application scenarios because of their two common properties:

- version control is essential for their functioning, and
- their users are inherently distributed over geographically distant locations

First, in Section 2.1, we briefly explain the basic background and terminology of peer-to-peer systems. Following that, in Sections 2.2 and 2.3, we describe Wiki and global software application scenarios and analyze the usage of the peer-to-peer paradigm in their communication. Section 2.4 summarizes the benefits of the peer-to-peer approach for those scenarios and weighs them against the drawbacks of centralized approaches. A running example described in Section 2.5 illustrates the use of a version control system, which is based on the peer-to-peer paradigm.

2.1 PEER-TO-PEER COMMUNICATION PARADIGM

Peer-to-Peer System

A Peer-to-peer system consists of one or more ***peer-to-peer overlays***, realized with communication ***protocols*** and one or more ***applications*** on top.

Peer-to-peer describes the communication, transactions, or exchange which occurs directly between equal individuals ("peers").

Peer-to-Peer Overlay Network

Peer-to-peer overlay represents a virtual network of peers built on top of the ISO/OSI-Layer 4 [Zim80], i.e., the transport layer, which is called the **underlay**. Each peer has a ***peer ID*** (also called overlay ID) and peers are connected by the virtual connections between each other. This virtual connections forms a so called overlay, which is based on physical routes in the underlaying transport medium.
A **peer ID** is a unique identifier that is sometimes related to the content/information a peer stores, depending on the kind of ***peer-to-peer protocol***.

This decentralization of the communication and data (e.g. messages, data or multimedia streams), without an intermediary server avoids a single point of failure, a bottleneck of communication, and reduces the maintenance and hardware costs. All participants of ***peer-to-peer systems*** have the same responsibilities and rights in its pure, original meaning.

Peer-to-Peer Application

After a direct connection is established a mechanism can exchange data with other peers. This effectively forms the application the user operates. This application can be further separated in **framework services**, which are reusable components such as data storage and **user level applications**, with which the user interacts directly. The peer-to-peer based Wiki engine presented in Section 9.3.3 is an example for a *user level application*, the version control mechanism, which is presented in Chapter 7 is a *framework service*.

Peers in a peer-to-peer system communicate with each other with a set of predefined messages, according to the particular peer-to-peer protocol [SE05]. In the ISO/OSI layer model,

a peer-to-peer protocol is built in the application layer, on top of the transport layer. It describes how peers find each other and communicate in a fully decentralized fashion. Peer-to-peer protocol messages include routing and maintenance messages. Peers have a unique identifier, the peer ID, and are connected to each other via (virtual) links. Such a network of peers with this communication protocol is called ***overlay***.

From its original meaning, peer-to-peer stands for communication between equals. There are, however, also centralized and hybrid, systems in addition to pure peer-to-peer overlays.

In a *centralized peer-to-peer overlay* one peer has a more important role. The indexing of content and finding which peer offers which ressource is performed by that peer, this single instance. The **index** is a list, which maps the ***identifiers*** to their respective resources. Content transfer is carried out in a decentralized fashion among peers in the overlay. A popular example of the centralized approach is Napster. In *hybrid overlays* there is more than one peer with a more important role, the so called *superpeers*. Peers are connected to their assigned superpeer and communicate with it in a client-server fashion, whereas communication between superpeers is pure peer-to-peer.

Peer Proximity

A peer proximity is the distance (based on, e.g., mathematical subtraction, logical xor) between a ***peer ID*** and a given ID (e.g. resource or other peer ID). How the **proximity** is calculated depends on the respective ***peer-to-peer protocol***.

Pure peer-to-peer overlays are further classified in structured, unstructured, and hierarchical overlays. In *structured overlays*, there is a relation between peer IDs and the resources they store and share. There is always a peer responsible for a part of an ***index*** of shared resources, and each peer is aware of that peer, through the peer/resource ID relation. Finding a resource means routing a request to the peer responsible for the part of the index referring to the required resource. Each peer has a routing table with peer IDs mapped to the transport layer address of a set peers, the so called ***neighbors***.

Neighborhood Peers

A peer A is called neighbor of a peer B, if B's ***ID*** is one of the n ***closest*** IDs among all online peers. A's overlay and ***underlay*** address is kept in the routing table of peer B.

Routing tables in structured overlays are organized in a way that enables so called *greedy routing*. That means that the routing is optimized to reach the destination in as few ***hops*** as possible. A **hop** denominates a peer who forwards a message. Each hop is chosen from peers that are ***closer*** to the destination.

An example of structured overlay is Pastry [RD01b].

Peer-to-Peer Protocol

A **peer-to-peer protocol** describes the ***maintenance mechanisms*** and ***routing mechanism***. The purpose of a peer-to-peer protocol is to establish a direct connection between any two peers.

maintenance mechanisms: Each peer knows a bundle of other peers in the network and keeps in contact with a few of them. For example, a peer can use probing messages, called **keep-alive messages**, to check if the peer from his routing table is still online. In the case that it is offline, the corresponding entry in the routing table is replaced according to the ***Peer-to-Peer Protocol***. Similarly, upon joining, a peer obtains a number of other peers' contact information. The mechanisms which take care of these operations are the **maintenance mechanisms**.

routing mechanism: Using routing mechanisms, any peer can be found. If the requested peer is unknown to the initiator of routing, neighbors are asked to forward the query until the destination is reached.

In unstructured overlays, there is no relation between a peer ID and the resources a peer's shares. A routing is mostly done by flooding – forwarding a message to neighbors who do

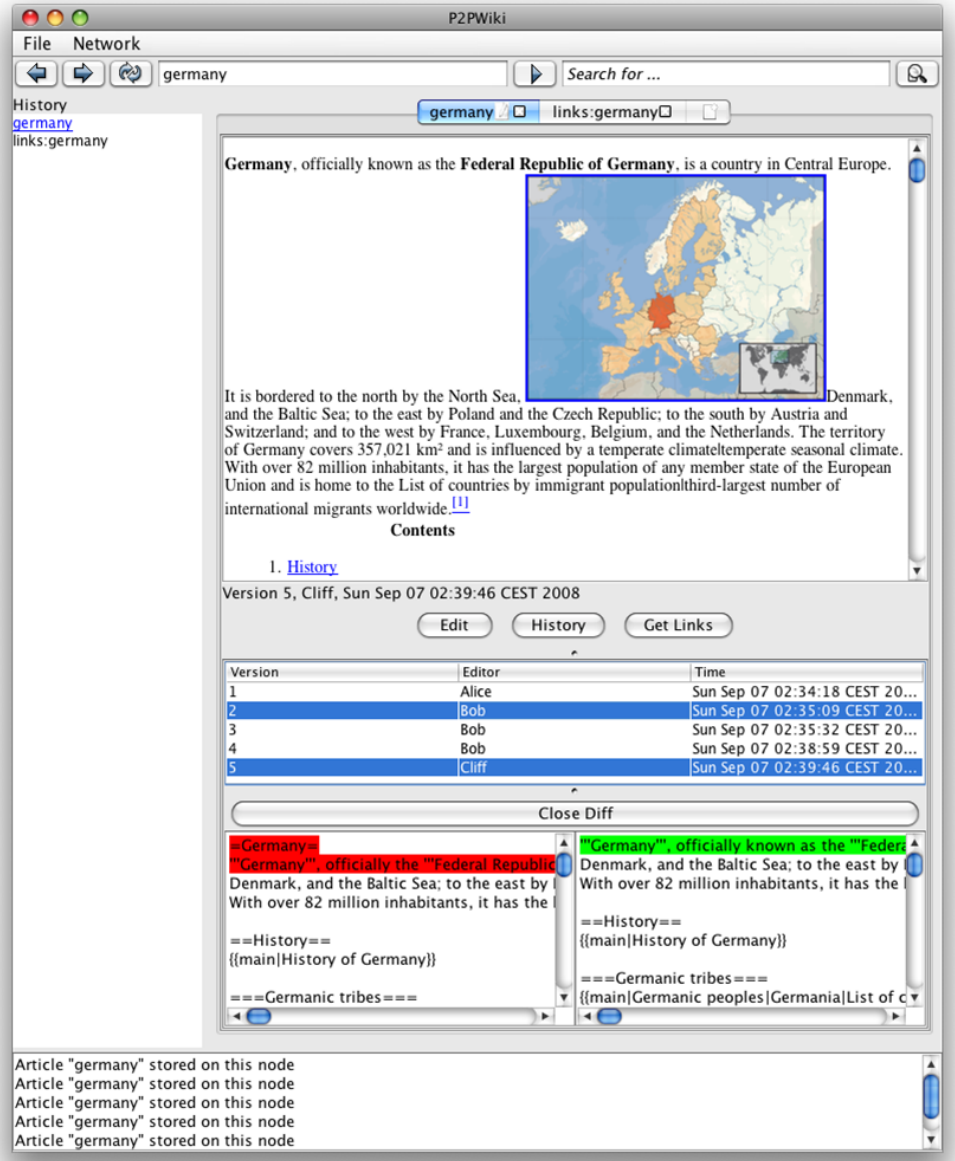


Figure 3: Peer to peer based Wiki engine Piki [MLS08]

the same until the destination is reached. This leads to indeterministic routing and often overloading the networking with the routing messages. There are, however, sophisticated approaches to minimize this overhead and achieve retrievability of the queries for the resources, such as in BubbleStorm [TKLB07].

All communication messages in peer-to-peer overlay, routing and maintenance of routing table, is defined by ***peer-to-peer protocol***.

Hierarchical overlays use the routing in one or more hierarchy levels. Peers are organized in clusters. Inside of the cluster one routing mechanism is used and clusters are connected by their cluster heads, reachable by their own routing mechanism. In this thesis we will focus on structured overlays.

2.2 WIKI ENGINES

One of the technologies which made the Internet more collaborative is the Wiki technology [LC01]. Ward Cunningham invented Wiki engines in 1994. It proved to be an appropriate

tool for sharing global knowledge, as it provides an easy way to contribute and consume information. Information is shared by writing a Wiki article, which is very similar to a normal webpage. It is edited using a simplified markup language (Wiki markup). Articles can contain other media such as pictures or sound samples and can be linked to other articles, creating a knowledge network.

The way Wikis are used, e.g., consuming and contributing information, is naturally suited to the ***peer-to-peer (p2p)*** paradigm, which inherently follows the structure of its distributed users.

There is no central point of communication, resources are used and offered by all users. The more users are using the system concurrently, the more capacity is required. When knowledge should be shared for free, as in the popular project Wikipedia [wikb], hosting costs should be as low as possible. An additional benefit is diminishing censorship as articles would be redundantly hosted by many distributed users. As a Wiki engine is intended to be easy to use, it is desirable that setting up and running a Wiki engine should not require extensive knowledge.

The crucial feature of a Wiki engine is its support for concurrent modifications of the stored articles. The contribution of an author should not unwillingly overwrite another contributor's changes, which may have happened concurrently. In order to understand the evolution of an article, all modifications should be retrievable in their executed order, preferably with the respective author noted and the ability to compare the changes made in any of these ***versions***. Although not provided by existing wiki engines, the ability to retrieve a linked article in the state it was in when it was linked would be helpful.

Figure 3 shows a peer-to-peer based Wiki engine based on the results presented in this work.

Existing Wiki engines are implemented in a lightweight manner, using script languages like php (e.g. Wikimedia, TikiWiki). Additionally, article versions are stored as database entries. They, therefore, typically also implement the functionality of ***version control systems***, instead of reusing a version control system for source code management.

Although not provided by existing Wiki engines, the ability to retrieve a linked article in the state it was in when it was linked would be helpful.

Version

A **version** is a modified copy of an artifact. All changes to the content of an artifact, which are stored using a ***version control system***, lead to a new copy of the respective artifact instead of overwriting it. These copies are *versions* of each other. The term version is distinguished further in **revision** and **variant**. A **revision** is an evolutionary change, an update of an artifact. A **variant** is a variation of the original artifact, e.g., in another language. The distinction is theoretical only, as revisions become variants, if they are based on the same artifact version. Looking at a line of modifications, which are based on each other, an artifact is a revision. While looking from the perspective of an alternative ***development line***, the same artifact is a variant. Thus revisions and variants are not distinguished further in most version control systems.

2.2.1 *Cooperation in a Wiki Scenario*

All articles of a Wiki engine are ***linked*** to each other. These links connect Wiki articles, which have semantically related content. The links should point to a specific version of an article, as updates could completely rewrite the article and change the semantics of its content. In practice, however, an update on an article's content improves the information in the article, without changing the entire statement of the article. Links always point to the latest versions and are never (automatically) checked, if the link is still justified.

Link

A **link** is a connection between ***artifacts***. Often this connection has a semantically meaning, i.e., two artifacts are connected, when their content is referring to each other. A link carries metadata about the connected artifacts. This excludes information about an individual artifact, but includes all information, which cannot be assigned to a single artifact alone. Therefore this information describes the connection between artifacts, e.g., the connected versions, the status of the connection, or constraints, with which the connection can be validated.

A link is not limited to connecting only two artifacts. Any number of artifacts can be connected. There is no default navigation direction of a link, although the direction can be specified as metadata.

Beside Wiki articles, there are also so-called *special pages*, which are presented like Wiki articles. However, these pages contain computed information only, as a list of all recently updated articles. These special pages often link to multiple articles.

A Wiki article becomes automatically linked to *special pages* and manually associated with other articles. Even if an article is not manually linked, it is linked to a special page. The connections of all articles in a Wiki form a connected graph. If these articles are not stored in one (version management) system these connections need to be broken up, creating two separate Wikis.

The articles' authors can be spread around the globe, depending on the Wiki's purpose. In the Wiki application scenario a similar usage to the Wikipedia project [wikb] is assumed. Here any person on the globe can participate. The interaction pattern, however, is only similar to the user behavior in the global software development scenario (shown in Section 2.3). Potentially any person could be interested in an article's content. The authoring group, however, is usually small, as only some people have the necessary specific knowledge to contribute to an article. This smaller group behaves much like software developers. Often a main author creates an article and maintains it by editorial updates. When others contribute to the article their contribution is most likely reviewed by the main author, or if not, by more contributors to the specific article. This interaction fits the peer-to-peer paradigm.

An important issue for a project, where knowledge is collected and offered globally, is censorship. If all information resides on a central instance, e.g., a server, it can be manipulated unnoticed easily. If access is granted or can be illegally gained all articles could be deleted or altered. Additionally, the access to this central source could be blocked by a network provider, who does not want their people to access the contained information, as some countries do today.

Distributed stored articles are stored with multiple copies to increase their availability. To manipulate an article unnoticed all of its copies would have to be modified. Because the peers, where these copies are stored, change dynamically it is not feasible to access them all unnoticed. Copies stored on machines, which are momentary offline, need to be accessed as well! Similarly, the access to the Wiki system cannot easily be hindered, if every participant can offer access to the complete system, as offered by peer-to-peer paradigm.

Additional benefits a decentralized solution would bring are low costs in setting up and maintaining hardware and software resources, in order to run the Wiki system. A peer-to-peer system operates solely on the participants' resources. There is no need for expensive server hardware. The maintenance costs of the hardware are reduced to the costs each participant has to invest to keep his machine operational, regardless of the peer-to-peer software. Software maintenance costs are settled in the same manner by the participants alone. The network connection of each machine has to be configured so that any participant of the Wiki system can be reached. The software, which implements the ***peer-to-peer protocol*** maintains itself, without the need for human intervention: Connections to other peers are maintained, data is downloaded and cached automatically, and the software could even update itself.

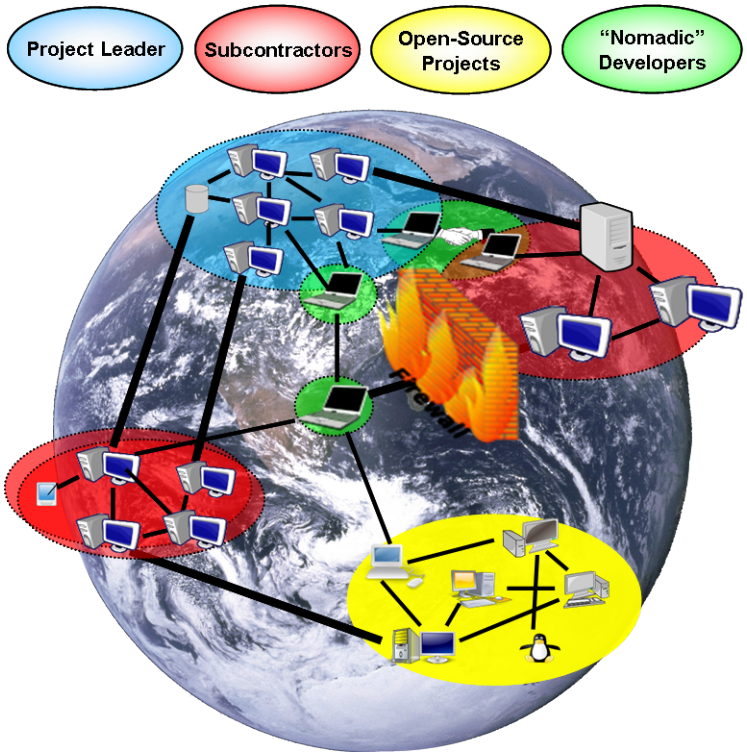


Figure 4: **P**eer-to-peer based **i**ntegrated **p**roject-support **e**nvironment (PIPE) [MKS08]

2.3 GLOBAL SOFTWARE DEVELOPMENT

A growing number of today's software projects are developed by globally distributed teams [MHK05]. The reasons for separating the development over different locations around the world are numerous. Outsourcing (motivated by involving experts in a field or simply to decrease cost) and efficient time management (using the advantage of different time zones) are some of them [MH01]. Although the following statements can be applied to the development process of any kind of project, we focus on software development.

Global software development (GSD) is typically organized as shown in Figure 4: The company leading the project is referred to as *project leader*. It employs multiple *subcontractors*. In IT-projects it is not unlikely that open-source software will be integrated so that the open-source developer community becomes part of the project. The developers of the mentioned parties work in different physical locations. Additionally, some workers, which we call **nomadic developers** travel between those locations.

In addition to cultural differences [PAE03], GSD must cope with a lack of appropriate support tools. The most widely used CASE tools are designed for on-site development where multiple clients are using one server, i.e., the client-server communication paradigm. Numerous field studies of GSD projects [HPB05, HM03, Šmi06] show that the biggest problem using these tools is the slow message transfer between physically distant machines. Recently developed tools aimed to support GSD, like Jazz [Jaz10, CHSJ03], still rely on client-server communication.

A centralization of the communication by a server is orthogonal to the natural structure of GSD. Developer teams normally manipulate the same artifacts and there will be some form of communication between them. This communication (including message exchange and file transfer) obviously does not need a remote server and should proceed directly from peer to peer. A peer-to-peer based approach is naturally suited for GSD as it is inherently distributed; there is no central point of communication. After a distributed routing algorithm connects the participants they communicate directly with each other.

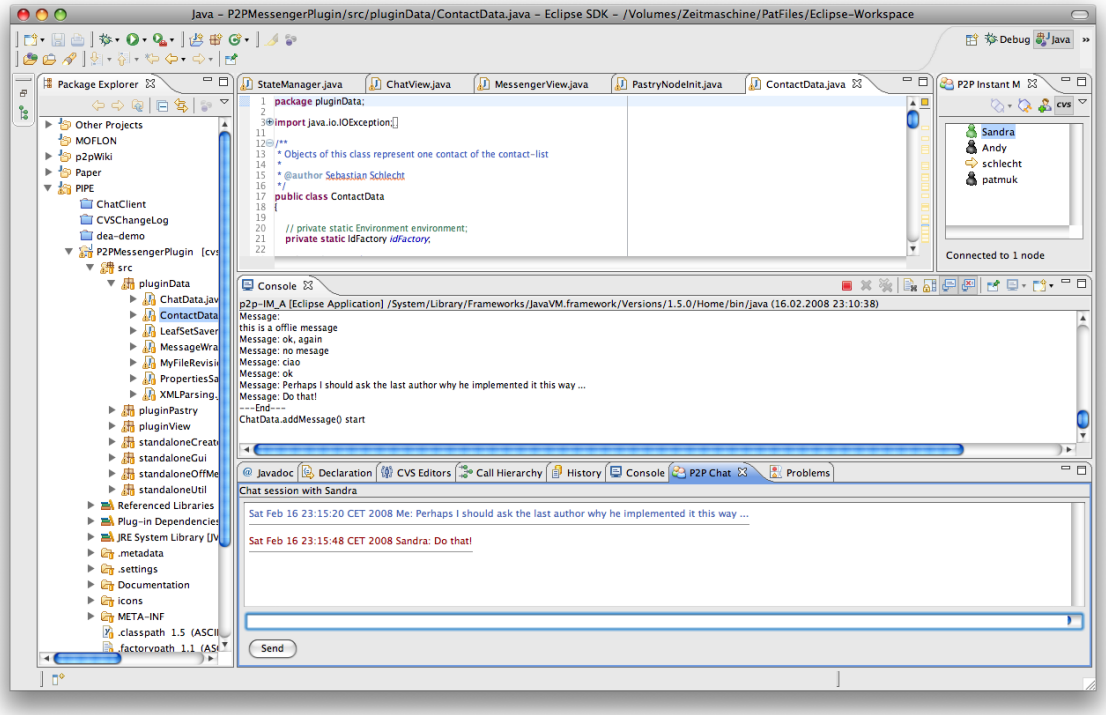


Figure 5: **P**eer-to-peer based **i**ntegrated **p**roject-support **e**nvironment (PIPE) [MKS08]

We can see a peer-to-peer based prototypical development environment, which is based on the results of this work, in Figure 5. It implements a basic set of features, being a version control system, access control, and an instant messenger, which highlights the last author of the currently opened document [MKS08].

Every software development process consists of different phases. The initial requirements engineering phase is followed by the design phase, in which the architecture of the desired software product is planned. Following this design, the work on the implementations begins, followed by the testing phase. In various software development processes, these phases are divided further, as well as the transition between these phases. Nevertheless all phases themselves are always present in every software development process.

We can define different roles, which usually relate to the development phases, e.g., requirement analysts, programmers, etc (see [Som10]). Usually there are additional roles, which also correspond to tasks not directly connected with the development of the software, such as management, or costumer support. These roles are, however, not all assigned to different people, for example the manager takes usually the role of a requirements analyst and designer as well. In a GSD project the cooperating persons work in different, often very distant, locations. Through the distribution of the persons working in the different roles, the development phases are practically distributed. The distribution can be both *vertically* (e.g. design in one site, development in another) and *horizontally* (e.g. multiple sites working in the development phase on different subsystems). Much like a concept of distributed computing, the project development should be carried out in as independent as possible sub-projects [HM03].

The evolution of the software development processes demonstrates that the development phases interact with each other. Artifacts produced in one phase are based on artifacts of former phases. Developments in later phases may change artifacts of earlier phases. This is especially the case when moving from the design to the implementation phase, or when the test phase influences the implementation phase. Therefore, development teams working in different phases have to interact with each other.

As there are no substitute for direct face-to-face communication [ISHGH07], a common practice in GSD is to let either developers or dedicated consultants, called ***nomadic developers***, travel between the project's sites.

Additionally or alternatively, software projects can be distributed by dividing the product's components among teams in different locations. It is good practice to couple them as loosely as possible (advised by [GHJV95]).

Components of a software project are connected through well-defined interfaces. Hereby the dependencies between the components are theoretically minimized, but experiences from real software projects showed that the interfaces are modified several times during the development process. In practice, the development teams of different components have to interact as well.

Supporting tools for GSD can have many different functionalities. Shared knowledge could be managed using a Wiki engine, which proved to be useful for collaborative requirements engineering as demonstrated by [HRH07]. In whichever way the development is distributed, the geographically separated teams have to interact with each other. The most important part of this interaction is the exchange of development artifacts.

It is crucial for any software engineering project to track all changes to the artifacts using a version control system. It provides a common pool, called repository, where all development ***artifacts*** are available to any user at any time. Artifacts are produced locally and shared with the help of a version control system, which communicates globally. The tools used to create these artifacts do not need to communicate between different sites. Although this could be helpful, for example, being able to edit an artifact with multiple authors at the same time, it is not necessary. A version management system should support the exchange of artifacts and keep track of modifications on them regardless which team of which company in which location is involved.

Artifact

Every document, which is produced by a tool in any phase of a development process is an **artifact**. These artifacts can have any binary or textual representation (compare to [UML09]). Modifications to the content of an artifact produce new ***versions*** of the same artifact. A specific version of an artifact, however, can be addressed by concatenating the version's identifier to the artifact's identifier (which is in most cases the artifact's name).

If cooperating teams are using their own server environments, significant synchronization problems may arise. These problems occur mainly in version control systems and may also affect access rights. In practice, these problems are ignored and ***artifacts*** are exchanged through numerous other channels like e-mail or USB flash memory drives. The solution proposed in this work aims to avoid exchanges with USB flash memory drives or e-mail.

Additionally to the stated problems the alternative of a centrally installed system raises the question of ownership, as one of the involved parties needs to maintain this system.

2.4 BENEFITS OF PEER-TO-PEER PARADIGM FOR THE APPLICATION SCENARIOS

A (distributed) version control system is crucial for Wiki engines and an integral part of a development environment, especially if the latter is used in a globally distributed developed project [Šmi06]. Additionally, a Wiki engine can enrich a GSD environment to provide a knowledge database or even as a lightweight collaborative requirement engineering tool, as shown in [GHHS07b, GHHR07b].

Both application scenarios have similar requirements, which are detailed in Section 3.2.

The majority of Wiki engines rely on a centralized version control system. GSD projects use centralized server based solutions as well. The centralization on a server is orthogonal to the natural distributed structure of Wiki engines and GSD. In a Wiki system every user is a possible contributor and consumer. Often a user contributes to an article she consumed earlier. Developer teams in GSD projects normally manipulate the same artifacts and communication will usually occur between them. This communication (including message exchange and file

transfer) obviously does not need a remote server and should proceed directly from peer to peer.

2.4.1 *Shortcomings of Server-centric Solutions*

A centralized approach to collaborative applications, like version control, has several drawbacks:

Single point of . . .:

- . . . *failure*: It lies in the nature of server based solutions that all services are offered from a single point. If this fails, the complete system is unusable.
- . . . *vulnerability*: If someone gains unauthorized access to the central server, he is in complete control over the offered service and stored data. This data could be stolen or modified, e.g., censored.
- . . . *ownership*: There is only one party which controls the central server by owning it. If offered services are to be used and, moreover, data is to be stored this party needs to be trustworthy. Especially when equal partners in a GSD project are working together, neither partner would like to rely on the other, or be responsible for the safety of the project server. Its failure and possible loss of data could lead to the failure of the entire project.

Fixed Location: The server resides in a fixed physical location. If the connectivity to this location is bad or slow, the offered service will perform poorly. Unlike on-site development, some developers in GSD are geographically distant from the server and, therefore, have to cope with delay in data transfer [HM03]. Distributing servers across different locations can only decrease the communication delay for all developers if all locations are covered. However, it leads to the problem of synchronizing these servers with it.

Does not scale with needs: If the offered service is suddenly used by an unexpected number of participants, the server may run out of resources, which might lead to a failure of the system, or at least some users will not be served. Additional capacities have to be bought and installed. Most of the time the full capacity remains unused. Unexpected events (like natural disasters or political assassination) can lead to a high interest of many users in the Wiki scenario. External factors can also influence the load on a GSD supporting server, although these can be more easily predicted.

Additional maintenance costs: Another important issue of the client-server approach are its high maintenance costs. In addition to the multiple client machines, which are maintained by their respective users, the machine that the system relies on, i.e., the server, have to be kept running. Software needs to be configured and updated, and hardware needs to be maintained and replaced when it wears down.

Additionally, resources (computation power, memory, bandwidth, and storage space) in the system are not optimally used: servers provide all their resources while resources on client machines remain unused most of the time. The cost of servers is especially problematic for non-profit projects (like open source projects) as they are normally founded by unreliable donations.

Multiple different systems cannot be joined: If cooperating teams worked with their own server based environments, they cannot merge their infrastructure easily. They would have to setup a new infrastructure for the planed cooperation. In practice, each team continues to work on their individual system and exchange the cooperative edited artifacts through numerous channels like e-mail or USB flash memory drives. Conflicts cannot be detected automatically and may be noticed a long time after work on the conflicting artifacts has been completed (e.g. when the final product should be integrated). If each team uses their own version control system, some artifacts exist in an identical version in both

systems - without being marked as identical. Apart from these significant synchronization problems, access rights might also be compromized.

To gain capacity and to soften the locality problem, often a cluster of servers is used. These servers can be distributed over different physical locations. They can offer copied services and data, or share the load by being responsible for a smaller part of the entire service/data, or by serving only users from specific areas (i.e. IP ranges). However, although extended, the offered capacity remains fixed and the servers limited to fixed locations. Therefore, the communication delay cannot be decreased for all users. Additionally, the servers still have to be administered from a central party.

2.4.2 *Characteristics of Peer-to-Peer based Solutions*

A distributed peer-to-peer based solution copes with the earlier mentioned drawbacks of server based solutions. The offered service is a distributed algorithm which runs on each participant's machine. Thereby, the user is offering services to other users and accessing services from other users at the same time. Data is stored distributed among multiple machines, with redundant copies to assure availability. The users' requests are distributed among multiple machines. The offered service exists as an identical copy on each machine, but might require the cooperation of multiple machines. The peer-to-peer routing algorithm takes care to connect two participants, who communicate afterwards directly with each other.

Robust against . . . :

. . . failure: A peer-to-peer system is designed to handle parts leaving the system. Any participant can leave at any time. His/her machine can even fail. Failing peers are detected and automatically replaced by their neighbors in the virtual overlay network. Data and service states are replicated prophylactically on these machines. The routing algorithm automatically directs requests to the replacing machine.

The number of neighbors, which replicate a machine's state, is configurable and called the replication factor. Only in the unlikely case that all of these machines fail simultaneously in a time window, which is smaller then the time needed to replicate the state to a new machine, the state and stored data are lost, until one of these machines reappears. The system is, however, still functional and only some data would be missing.

When a huge number of participants fail, the routing algorithm might not work as it should due to missing routing information. The network could split into loosely connected or completely disconnected partitions. The system would be functional in each partition, but some data might be missing.

. . . central control: As the data is randomly and dynamically distributed among the participants the control of a single participant, does not gain access to all stored data. Data cannot be modified unnoticed, because it exists as identical copies on a number of replica.

Collective ownership: There is no single owner of a peer-to-peer system, as it utilizes the resources every participants offers. A central control is difficult to achieve.

Omnipresent: A peer-to-peer system is available on all participating machines at the same time. In the best case, the requested resources are offered by the one's own machine, in the worst case the offering machine is far away. Due to the dynamic changes in the network caused by leaving and joining peers, the actual communication delay to an offered resource changes, since leaving peers are replaced by other peers. Peers are uniformly distributed, thus the delay can get smaller, if a physical closer peer takes over, or worse, if a distant peer replaces the leaving peer. In a high dynamic environment, where peers frequently leave and join, the experienced communication delay over a

longer time is approximately the average delay among all machines. However, if peers stay long times in the network, and/or don't change their identification when joining, the communication delay is always the same.

When peers in one physical location leave their offered resources are replicated and offered among the peers left in the system. In a system, which consists of machines in two physical locations, shifted online times due to different time zones have a positive side effect we call **shifting**, where the resources offered in the distant location are automatically shifted to the peers in the other location, reducing the communication delay to access those resources for the peers in the same location.

There are different caching mechanisms (e.g. owner path caching [CSWH00])that can lower the experienced communication delay.

However, routing to a peer offering a specific resource takes a remarkable time as well. If the peer has not been contacted recently, a lookup message has to be sent to a number of peers in order to find the peer offering the needed resource. Subsequent messages can be sent directly using the underlying network's address.

Scalable: The strongest advantage of peer-to-peer systems is their inherent ability to grow in capacity with a growing number of participants, as each participant's resources are utilized. Fast growth stresses the system nevertheless, as it needs time to reconfigure itself to the new number of participants, which involves updating of routing contacts and redistributing stored data. A sudden interest from a large number of participants to retrieve the same data also stresses the system, as the requests are unevenly distributed and ask for a service offered only by a small number of the participants.

Minimal maintenance costs: Cost of ownership and maintenance costs are very low. As in client-server environments each individual machine has to be maintained by its user, however, there is no machine in addition to these machines, which needs to be maintained. Thus there is no cost to set up the system other than to start each individual machine and contact an already connected machine.

The dynamic changes in the system are handled without user intervention by maintenance and failure handling algorithms. A failing system is the expected standard behavior, called ***churn*** (see [SR06, HT07]).

Churn

The dynamic behavior of a peer to go offline and online again is called **churn**. A peer can **leave** the system gracefully, i.e., announcing its departure to give its ***neighbors*** opportunity to take over its duties, or simply **fail**. In the latter case an automatic mechanism detects the absence of the peer and a neighbor takes over its duties.

Although not strictly limited to, churn refers to the case, where a number of peers are ***failing***. The behavior of peers using file sharing applications has been observed by [SR06] and can be described with a Weibull distribution. Further details are discussed in Section 10.3.4.

As long as one machine is running, the system is functional, although a large number of simultaneously failing peers might seriously harm the systems behavior.

Disjunct systems are joined automatically: If coexisting peer-to-peer systems, which offer the same service, should be joined to form one system, no additional overhead is necessary. If one machine contacts participants of both systems the systems join each other and synchronize automatically. Form the point of view of each system this situation is identical with a large number of joining machines.

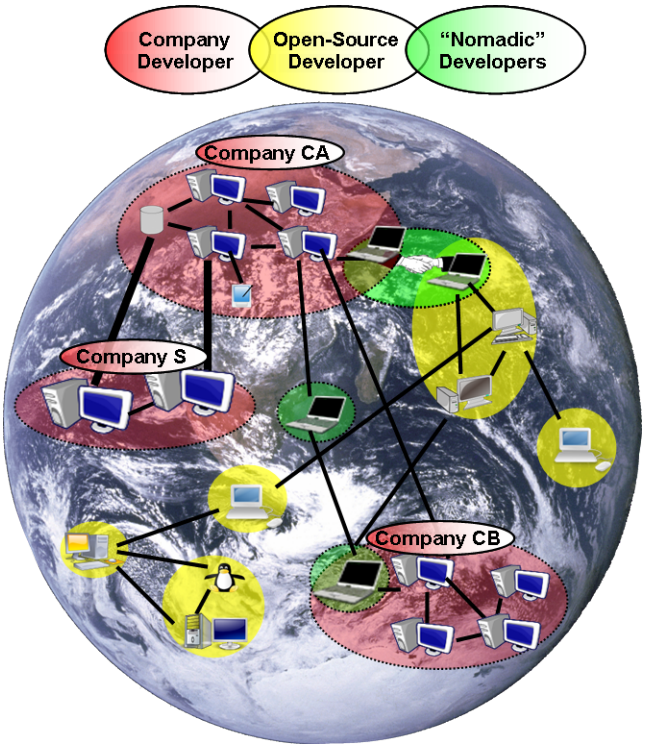


Figure 6: Globally distributed developers in a GSD scenario

2.5 RUNNING EXAMPLE

For a better understanding, a running example is outlined in the following section. Whenever a mechanism is explained the example used will be based on this background story.

The proposed version control system will be most usable in a GSD environment, as described in Section 2.3. In a project with globally distributed partners, a software product is created. The outlined scenario is depicted in Figure 6.

The outlined example is inspired by the development process of the open source software project Eclipse. As in any open source project, a number of independent programmers develop the product, mainly in their free time. Additionally, some companies are involved, which work full-time to improve the product. Next to these developers, some companies solely focus on the development of components, which they often sell as their product. As these optional extensions are dependent on the open source product they are involved in the core development as well.

In the outlined example we have four parties. A client company, called the stakeholder or short *S*, needs a software product, which fulfills a specified functionality. The company *Berlin Consulting*, shortened to company A or *CA* in the following, was hired to create this product. As experts in the domain the client is in – let us assume it is the banking business – they interact directly with *S*. Both companies are located in Germany, *S* is located in the city of Frankfurt while *CA* operates from Berlin. Meetings with the client are usually held in Frankfurt. In order to cut costs they subcontracted a company specialized in coding, which resides in India, Kolkata. The company name is *Calcutta Coding*, which we will refer to as *CB* in the following. *CA* and *CB* interact often and directly, while *S* interacts with *CB* only indirectly by exchanging artifacts. The planned software product consists of components, which are partially developed by open source projects. Open source components are often included in commercial products, in the form of libraries. In this example, an open source component is an integral part of the product. The fourth involved party are, therefore, the numerous open source developers, which are spread around the globe. None of them are active at all times, and some of them are only active for a period of time, without continuing

their participation ever. These independent and often anonymous developers are referred to as *OS*. The behavior of these open source developers is unpredictable. Their number varies, as does their working time. Their availability cannot be relied upon.

During the progress of the project, participants from the different groups have to meet in person [ISHGH07]. In doing so, the complete working environment with all artifacts should become available at the location where the meeting takes place. Changes made in another group's location should be trackable in the usual environment, thus eliminating the need to maintain identical copies in different systems manually.

To personalize the examples in this work, the following workers are selected: Alice is a designer and developer of company *CA*, Bob is her counterpart in company *CB*.

2.6 SUMMARY

This chapter argued and gave reasons as to why traditional version control systems, which are based on the client-server communication paradigm, are not useful for applications whose users are spread around the globe. The listed shortcomings of client-server based solutions and the benefits that the peer-to-peer paradigm can bring apply to any distributed applications. Statements focused on the drawbacks of existing solutions are detailed in Chapter 5. The benefits the peer-to-peer based solution presented in this work brings are detailed in Section 2.4.

Two application scenarios are laid out, where version control is crucial: Wikis and global software development. The more demanding scenario is the global software development, which covers most of the requirements derived from the Wiki scenario. Additionally, the Wiki scenario can be combined within the global software development scenario.

A running example, which is used throughout this work, details a use story, where both application scenarios are integrated. The focus of this example and the presented solution, however, lies in global software development.

3 REQUIREMENTS AND ASSUMPTIONS

In the previous chapter we explored application scenarios where version control is a vital functional part and analyzed the suitability of the peer-to-peer communication for these scenarios. That provides us with a basis for defining a complete set of requirements for a peer-to-peer version control system that fully meets the needs of collaborative development nowadays. In Section 3.1 we list the assumptions for the design of the final solution. Functional and non-functional requirements, derived from the elaborated application scenarios and limitations of used technologies, are presented in Section 3.2.

3.1 ASSUMPTIONS

The goal of this thesis is to present a distributed version control system that supports both presented application scenarios adequately. Their needs differ, but complement each other when combined. Both application scenarios need a system to keep track of changes to stored data. Concurrent modifications have to be supported. While the wiki scenario only requires a focus on single files, the GSD scenario demands the control of modifications on multiple files, which should be tracked as one event (e.g. ***snapshot*** version control).

A user in the wiki scenario is usually not interested in all existing articles, whereas a developer in the GSD scenario needs updates on all files. Variant management of artifacts is needed in a GSD environment as well as in the wiki scenario, where a variant could represent the same article in different languages. A common practice is, however, to have an isolated system for each language. Nevertheless, if a version control system can handle the more strict requirements of GSD, the requirements of a distributed wiki engine are also fulfilled.

Snapshot

The momentary state of all files in a specified folder in a file system is recorded in a **snapshot**. Thereby, not only the modification on single files but the state of other files at this moment is recorded as well. Every snapshot taken represents the actual state of a user's files at a specific time.

In Subversion, for example, it can be that a snapshot is a mixture of the modifications of different users on different files: if one user always changes one file, and another user changes another file, and both are committing their changes without updating their working copy first, the recorded snapshot includes the changes of both files merged, instead of only one file, as in the user's working copy.

All snapshots are recorded in the order they are based on each other in the **history**.

The behavior of the participants in the GSD scenario is more predictable than the users' behavior in the wiki scenario. They tend to use the version control services in a repetetive pattern. They are clustered in globally distributed locations, while the wiki users are distributed randomly. They tend to stay longer in the system and leave at predictable times compared to the unpredictable access of a wiki user, who usually leaves once his needs are served. By supporting the more dynamic behavior of a wiki engine user a GSD developer can be supported as well. Open source developers, however, behave similarly to wiki users, as their participation is less predictable.

With the aforementioned assumption that the described GSD scenario's requirements comprise the wiki scenario's needs, the proposed peer-to-peer version control system focuses on the GSD scenario.

The decentralized, peer-to-peer based version control system presented in this work is based on the following assumptions:

A-1: Network connection: A user needs to have an IP connection to other participants of the system. This connection, however, does not have to be permanent while using the system. It is only necessary to exchange committed versions with other users. All other system services are functional without any network connection.

A-2: No heterogeneity: All participating machines are treated equally. The system requirements of our prototype are, however, low enough that a typical development computer should be able to fulfill the most demanding tasks.

A-3: No dedicated single point of control: To overcome the shortcomings of existing solutions (mentioned in Chapter 5) our solution is designed to operate without a central, irreplaceable instance. This fact, however, does not limit the retrievability of all controlled files available.

A-4: Unreliable user participation: Each participant can leave the system at any time.

A-5: Users are located anywhere on the globe: A user can participate from anywhere, where he can reach any other participant via network connection. Users can be physically close to each other or widely spread around the globe.

A-6: File format: Our prototype handles any kind of files. Text files, however, are more effectively supported, as ***delta compression*** can be aplied instead ***binary delta compression*** to efficiently store the versions of a file.

3.2 REQUIREMENTS

The following requirements are derived from investigations of the very popular wiki project wikipedia [wikb] and selected industrial field studies on GSD [HPB05, HM03, PAP06, Šmi06, Sou01, ISHGH07]. Additional requirements emerge from the usage of peer-to-peer technology. The requirements are presented as proposed by [RR06] divided into non-/ and functional requirements. The last section contains security considerations we are aware of. However, meeting them is not within the scope of the presented version control system. All requirements are listed in the comparison overview in Table 1 in Chapter 6.

3.2.1 *Functional*

Functional requirements are directly visible to the user as the system's features. They can often be triggered directly or indirectly by the system's offered services.

R-1: Support common version control operations: A minimum set of operations, common to almost all version control systems, should be supported. The following requirements state them:

- ***R-1.1: Support parallel work on all artifacts:*** Developers should not be block each others work when using a version control system. They should be able to work on the same artifacts at the same time. The version control system deals with contradictory changes, which is covered by requirement *R-1.2* and requirement *R-1.3*.
- ***R-1.2: Detect concurrent modifications:*** Modifications to the same artifact by different users should not go unnoticed. Users expect their contribution to be the only one in existence, but whenever the same base version is changed, concurrent versions are created. Some systems (mainly the ***centralized version control systems***) prevent a concurrent version from being committed - the conflicting version's changes have to be integrated first. Some systems (mainly the ***distributed version control systems***) allow concurrent versions to exist in the repository as parallel versions, but encourage to merge parallel versions into a new version. A user should always be notified about concurrent modifications that occur when concurrent changes are based on the same version or snapshot. Regardless of whether the modifications

are conflicting, or if different artifacts in a snapshot were changed, the user should be informed. Merging concurrent modifications automatically does not guarantee the prevention of errors, as some dependencies might exist between different parts of an artifact or different artifacts. All recorded versions in a repository should be acknowledged by a user, not being created automatically.

R-1.3: Resolve concurrent modifications: The user should be assisted in resolving conflicting concurrent modifications. If the modifications are not conflicting with each other, i.e., contradictory changes in the same line of the same file, the changes could be automatically merged - but the user should revise this, as only he is able to understand the semantic context of the artifacts involved. The system has to show the user how concurrent modifications differ, so that the user can merge them to a new snapshot by choosing which modifications should be applied.

R-1.4: Track single artifacts in an ordered history: Changes to single artifacts should be recorded in the sequence in which they evolved. Whenever the content of an artifact is modified a new version is created. The version is **based** on the original artifact. A predecessor/successor relationship between the created versions could either be explicitly stored or computed using the relationship between the snapshots which the changes are part of.

R-1.5: Track snapshots in an ordered history: Changes to a set of artifacts should be tracked in a version history. This history stores the additional information, in which it states the state other artifacts were in, when a specific artifact was in a specific state. Tracking the state of single files only, as demanded by requirement *R-1.4*, could not answer which version of a related artifact is related to a chosen version of another artifact. All modifications of all changed files are recorded in a **changeset**. Adding the implicit information, which files were not changed, forms a ***snapshot***. As pointed out in requirement *R-1.4* multiple snapshot should be stored in an ordered history to represent their evolution.

R-1.6: Retrieve a recorded state: A snapshot represents the state of all artifacts in a project of a specific user at the recorded time. Thus, even if the user did not update his working copy to integrate the latest changes, a new snapshot recording the user's latest modifications should be an exact copy of her working copy (without incorporating the unintegrated changes submitted earlier by other users).

R-1.7: Choose recorded versions of selected artifacts: Similar to the normal behavior when single artifact versions are tracked, versions of selected artifacts only should be retrievable. This behavior is needed in a system, which records snapshots of a project as demanded by requirement *R-1.5*. The ability to pick specific versions of only some artifacts from recorded snapshots, called **cherry picking**, is helpful, whenever diverging ***development lines*** are to be combined. If the development is distributed among different teams, which change some files, their contribution needs to be extracted in order to integrate all modifications into a new snapshot, forming a final product. Especially in ***GSD*** being able to do this is helpful.

A user of a wiki engine may not be interested in all artifacts (or articles in this case) but a subset, which can consists of an article with its meta information and maybe a related discussion page. The directly semantically connected artifacts (e.g. a discussion page or other meta information) should be retrieved in a matching version - they would be useless otherwise. If such artifacts exist the supporting version control system must be able to record this relation.

Having a system, which fulfills requirement *R-1.5* and requirement *R-1.7* would enable the retrieving of coupled artifacts in a matching state, without the need to retrieve all existing artifacts.

R-1.8: Individual working copies: Each developer should have his own working copy, where he can modify artifacts without sharing his modifications immediately with

other developers. Initiating the *commit* operation records the current state of the files to form new versions. In her working copy only the user manipulates files, and decides which updates from the repository to integrate.

R-1.9: Enable local commits: Modifications, which are not final, should not be shared with other developers. They bring the project into an unstable state, e.g., hindering compilation of a software project. Those modifications would only harm the other user's progress. Having a working copy a developer can postpone to share her changes to the artifacts - but to be able to restore older states in a more fine grained way, e.g., to correct mistakes, it would be better to have a two stage commit process. First changes should be recorded only locally without sharing them, and later a user can decide to push them so that other users can access her new versions.

R-1.10: Track variants (branches): Other than evolutionary changes, which are based on each other, variant changes should be tracked as well. ***Variants*** are based on the same version and coexist in separate branches. In contrast to conflicting ***revisions*** variants are not meant to be merged into a new version. However, changes made to one variant might have to be integrated to the other variant. A version control system has to support variants, usually by offering ***branches***.

R-1.11: Tag with labels: When developing a software product it is helpful to mark certain states of the project with custom labels. The abstract internal version numbers which version control systems assign to recorded changes can hardly be memorized by a user. A tagging mechanism should support labeling specific snapshots with tags such as with "release 1.0" or similar user friendly names.

R-2: Change the name or path of artifacts: The name or the path of artifacts should be changeable, without losing the recorded history.

R-3: Support for multiple projects: When globally distributed companies collaborate the used version control system should be able to handle multiple projects in individual modules of the same repository. This avoids organizational overhead from managing multiple systems. The ability to partition the entire repository into projects might be useful for the wiki scenario as well. Categories of articles or articles for different countries in different languages could be organized in separate projects.

R-4: Conjoined local repositories: A study on nine GSD projects [HPB05] showed that each site had its own version control server so as to avoid the constant delay introduced by using a common one. The produced artifacts, therefore, had to be synchronized, which involved manual work where errors occurred. In order to avoid these problems the system should offer a repository at each site, which is connected and synchronized with all other repositories.

R-5: Redundant backups: Storing all states an artifact has ever been in serves two purposes: Sharing them with other developers, who can understand evolutionary changes, and conserving artifacts, if they are needed at a later time. The latter purpose offers a backup functionality. Version control systems are not designed initially to offer a reliable backup function. Some (e.g. the ***distributed version control systems***) completely omit this feature, some (any ***centralized version control system***) can be extended to enable backups and others offer reliable backups as a side effect of their architecture (e.g. many ***peer-to-peer based version control systems***).

R-6: Interoperability of different VCSs: Sometimes different parties are unwilling to switch to the same new version control tool. Porting all data to a new environment might by impractical in a running project. The global version control system should be able to integrate other systems to support a heterogeneous tool environment. Thus, existing systems could be merged, using the global version control tool, automatically.

R-7: Connect from anywhere: The tool environment should allow any developer to take the role of a nomadic developer at any time. Nomadic developers are assumed to work in different places, even during their travels. Wherever a user can get access to the Internet he should be able to access the system. Specific network setups, such as ***NAT***, can prevent this.

Network Address Translation (NAT)
A router maps a single external IP address to multiple internal IP addresses, so that internal hosts are able to share an external address. In this setup a connection to an external host can only be initialized by an internal host.

R-8: Offline version control: Nomadic users, in particular, need to be able to work offline (e.g. during travels). Modified artifacts should be transparently synchronized as soon as the user comes online.

R-9: Local commits: It should be possible to commit changes locally only so that they can be shared globally at a later time. With this functionality fine grained changes, which are not yet intended to be seen by other developers, can be kept locally, until a stable version is created. Having this stable version all locally recorded changes can be shared in one step. A more comprehensive motivation for this requirement can be found in Section 4.2.

R-10: Traceability: Artifacts may contain links to other semantically related artifacts. It should always be possible to trace and locate those linked artifacts in the system. Traceable links are used in a wiki engine to navigate from one article to another, but would be helpful in a software development environment as well to trace the development through different phases and different resulting artifacts or to record dependencies for the build process.

R-11: Easy to setup/manage: The system should be as easy as possible to handle. If the setup and management of the version control system is complicated, the user group is limited to the ones with enough knowledge needed to handle the system. The harder and more time intensive the installation of a tool before its use is, the more unlikely it will be used. Additionally, an incorrect configuration is more likely to occur. The system should enable, e.g., a small number of developers to spontaneously meet for a small project, where a low startup time is important.

R-12: Prevent censorship: Once any information resides in the system it should be not deletable. Artifacts can be marked as deleted, but have to be still stored, to be retrievable any time as postulated by requirement *R-1.6: Retrieve a recorded state*. It could be in the interest of a participant to remove information which was inserted by mistake. This, however, should be accomplished by setting access rights (if access control is provided as well). Allowing deletion would prevent requirement *R-1.6: Retrieve a recorded state* to be fulfilled.

Additionally, unnoticed manipulation of stored artifacts should be impossible. Any modifications should be traceable to its author and revertible by retrieving its parent version. Tracing changes could be used in a wiki engine to repair vandalism or censorship (i.e. by finding all changes made by a specific author).

3.2.2 *Non-functional*

Non-functional requirements are not directly visible to a user. They describe *how* the system should behave while fulfilling the functional requirements stated before.

R-13: ACID: Every transactional system should fulfill the ACID properties introduced in [HR83]. **ACID** is an acronym for:

R-13.1: Atomicity: Every transaction item, i.e., all changes to all artifacts a user wants to commit at once, should be applied or none. No partial changes are allowed to be stored in the system.

R-13.2: Consistency: The repository should be in a ***consistent*** state after a transaction. A consistent state is present, when specified integrity constraints hold. These constraints are stated in Section 4.4.2. If ***coherency*** is not guaranteed consistency cannot be provided.

R-13.3: Isolation: Transactions should not influence each other. The result of concurrent commits should be the same, no matter in which order they were applied. Conflicts should be also detected, regardless of the order the conflicting snapshots were committed.

R-13.4: Durability: The result of a transaction should be stored permanently. Even after a (partial or full) system failure all committed artifacts should be retrievable. Partial system failure, i.e., peers leaving the system, is the normal case in a peer-to-peer system, as every participant can leave any time.

R-14: Availability: There should be no constraints to accessing the system due to the network a participant is connected to, time of access, or working platforms (e.g. OS). Note that the demand to be always able to access stored data is covered by requirement *R-13.4: Durability*, while network conditions are covered by requirement *R-7: Connect from anywhere*.

R-15: Transparency: It should be transparent to the user how the distributed system works. She should not need to worry about where to find the offered services, how to access them or where the data is stored. The distributed system should behave as if everything runs on the local machine. This is stated as a general design goal for distributed systems in [CDK05].

R-16: Minimal storage space consumption: The stored changes should consume as little local storage space as possible. To fulfill this requirement, typically ***delta compression*** is used. A positive side effect is reduction of the communication bandwidth consumption while transferring changes.

delta compression

Delta compression is an efficient way to store all versions of an artifact in a single location. By storing only the differences between two versions, called a **diff**, only the actual changes are recorded, instead of the full artifact that forms a new version. The delta can be computed using either the previous (**backward delta**) or following version (**forward delta**). Starting from a complete version any version can be reconstructed by applying the deltas between those versions. For a faster retrieval of versions specific versions are saved in their full form, consuming more storage space.

The ***diff*** is most accurate when calculated using text files; here only the differing text lines are stored. A diff can also be calculated on the bases of a binary representation of a file, called a **binary diff**. In the so called **binary delta compression** the differing bits are stored instead the full version of a file. However, the amount of changes and the size of the resulting diff are not correlated.

R-17: No centralized services: In order to avoid the shortcomings described in Section 2.4.1 the version control system should not rely on centralized services, even if these services are not bound to a specific machine, but are offered by a single peer when it is online.

R-18: Adequate communication speed: The delay of communication and transfer in the system should be minimal. Difficulties introduced by communication delays were observed in the field studies in [HPB05] and [HM03] and concluded to be critical to the project success. In these field studies client-server based version control systems were used.

R-19: Scalable and robust: A system should be able to tolerate the dynamic participation of its users, and not fail or degrade its performance.

Robustness

The **robustness** of a system shows itself, when a large failing part of it becomes unavailable in a small amount of time. It is measured by the degree its operations are affected. When, for example, a turbine of an airplane fails, a robust airplane can continue its flight without disturbance. A peer-to-peer application can show itself robust, when a large number of its users are failing.

The **scalability**, in opposite, shows how a system performs under an increasing ***load*** coming from a growing number of users.

R-19.1: Robust to highly fluctuating users: In open source projects, as well as in the wiki scenario, the number of participants fluctuates. Their behavior is hard to predict. There can be very few participants in one moment and a huge group of participants in the next moment. In the wiki scenario we can assume that a popular event draws attention to a large number of users, who want to participate in the system. A couple of hours later the information about that event could be unimportant again, leading to a shrinking number of participants.

R-19.2: Robust to shifting users: Participants of GSD usually join and leave a system at specific working times. But due to different time zones, it is likely that a large number of participants from one area will leave and participants from another will join within a short time frame. This sudden ***shift*** in the geographical distribution of the participants should not influence a system's performance. Replication and routing information update mechanisms of peer-to-peer overlay networks are crucial here.

R-19.3: Scalable to a growing number of users: The system should be able to support a growing number of participants without decreasing performance. With rising popularity of a wiki engine or open source project, or the rising importance of a software project, where more developers are assigned, the resource demand of the system increase. Even if the increase is sudden, the system should be able to handle it. A decreasing number of users is covered by requirement *R-19.1*.

System Load

The load of a system quantifies how exhaustive the services offered by a system are used. The load can be measured by numerous ***metrics***, one of them is the size of the transferred messages.

R-20: Continuous expendability: The system should not be stopped for an upgrade to the software or the hardware. That means that there is no unique hardware part in the system which cannot be switched off for an upgrade. It might be impossible to upgrade all distributed running instances of the software. Thus more recent versions have to be compatible with older versions. This is one of the design goals for distributed systems described in [CDK05].

R-21: Fault tolerance: A globally operating version control system should be able to recover form any kind of failure. Hardware, as well as software, might fail, messages might get lost or resources might vanish.

3.2.3 *Security Aspects*

A highly distributed system, where confidential data is shared between business partners, requires sophisticated security mechanisms. Many new security challenges arise when the tasks that are usually fulfilled by a single trusted node are divided among multiple entities [Wal02]. In the following, we will describe the security requirements we deem most important for a GSD environment. However, focusing on the challenges version control brings with it, we only developed a rudimentary security system, described in Section 9.2.2.

R-22: Access and usage control: Access control plays an important role in a software development process. Confidential documents have to be protected from unauthorized access without impairing the overall system efficiency. In a GSD environment, selected group administrators are responsible for defining security policies for their respective user groups. The participants have the option to further restrict access to their files. A peer-to-peer approach introduces even more problems, since there are now multiple entities responsible for defining security policies for their respective groups. Enforcement of those policies has to be deferred to the participant [SZRC06], since availability of central authorities cannot be guaranteed.

R-23: Keep sensitive artifacts: Companies might be interested to prevent sensitive artifacts to be copied on machines, which do not belong to themselves but collaboration partners. Even encrypting the stored artifacts using an access control, as demanded by requirement *R-22: Access and usage control*, might not be sufficient to prevent unauthorized access. Not giving physical access to those artifacts is the safest means of protection. This demands control where specific artifacts in a distributed system are stored. To fulfill this requirement in a peer-to-peer based system is extremely demanding, as artifacts are typically stored randomly distributed among all participating machines.

R-24: Attribute based access control: In a large software development process, it is often unnecessary to control the access on an individual basis. For most tasks it suffices for participants to be identified via their respective attributes. An administrator can assign a signed certificate to a group member, certifying his developer status. Other participants can then base their access decision solely on the presented user credentials. Park et al., for example, developed a role-based access control approach for a collaborative enterprise in peer-to-peer computing environments [PH03].

R-25: Confidentiality: Stored artifacts should only be accessible to authorized users. Under no circumstances should a user without the needed access rights be able to read or modify stored artifacts.

R-26: Data integrity: Forged artifacts have to be recognizable as being manipulated. Even manipulation in a version control systems history or other metadata should not go unnoticed.

R-27: Non-repudiation: Changes made should be traceable to their author. It should be provable that the changes were applied by the author.

R-28: Authentication: The aforementioned security goals such as confidentiality, data integrity, access control and non-repudiation depend on proper authentication. If the system for user identification fails, the mentioned security goals cannot be met in a satisfactory manner. On top of that, it is not always easy to decide whom to trust in a highly distributed system.

R-29: Secure communication: The communication between the participants needs to be confidential and data integrity needs to be guaranteed. This requirement is tightly coupled with the previous requirement. Efficient and secure group communication mechanisms have to be provided by the development environment.

R-30: Low administration overhead: The administration overhead introduced by additional security mechanisms has to be kept as low as possible. Software developers participating in the system should not have to deal with defining specific access rights for their files. Though this should be transparent to them, it should be the task of a group administrator.

3.3 SUMMARY

In this chapter we derived and analyzed a complete set of requirements for a peer-to-peer version control system that fully meets the needs of today's collaborative development. A list

of assumptions along with the exhaustive requirements listed in this chapter give a clue as to the general applicability of the solution presented in this work.

Part II

VERSION CONTROL SYSTEMS

After concretizing the changed needs of today's project development, we investigate current solutions in the coming chapters.
We begin by defining the basic characteristics all version control systems share in Chapter 4. The quality of a version control system shows itself in the degree of consistency it provides, which is also defined in this chapter. Classified into centralized, decentralized and peer-to-peer based solutions we present representative version control systems in Chapter 5. Their strengths and weaknesses are analyzed in Chapter 6, where we conclude the impact of design decisions on a system's properties.

4 FOUNDATIONS OF VERSION CONTROL SYSTEMS

This chapter presents the background information about the foundations of version control systems. First the basic collaboration workflow on how developers update their files and share their changes is described.

4.1 COLLABORATION WORKFLOWS

A workflow defines how the contributors of a project work together. When using a ***centralized version control system*** there is only one possible workflow: the centralized workflow, which is shown in Figure 7. Artifacts are changed in a working copy and shared with colleagues by committing them to a centralized repository. If the modified artifacts conflict with changes, which occurred in the meantime, they are first merged locally with the latest snapshot from the centralized repository, then committed. Merging changes can be postponed by working on a branch which is merged once the tasks is fulfilled.

This workflow, however, is dominated by the applied development process, which defines which tasks (i.e. design, development, integrating, testing, etc.) are executed in which order and/or by whom (in different roles including designer, developer, integrator, tester, etc.). To divide the project between different teams and/or tasks, branches are used (i.e. a maintenance branch for bug fixes on the delivered version, branches, where a new feature is developed, etc.).

When a ***distributed version control system*** is used, the members of a project have to agree on a collaboration workflow, to exchange their developments in a structured order. This collaboration workflow and the applied software development process have to fit to each other, and not every collaboration workflow fits every development process.

There is no mechanism in a distributed version control system to notify its users about new versions. To announce a contribution, like a bug fix or a realized feature, mailing lists are used, whereby a contributor posts his contribution along with details how to pull the contributions from him. There are some participants who took on the role of an integration manager, who chooses from the different contributions in order to form a "blessed" repository, which presents the official acknowledged history of a project. This repository if often published on a central server, like gitHub[1], to offer read only access to every interested user.

Some prominent workflows are detailed in the following.

Centralized

The **centralized collaboration workflow**, which is illustrated by Figure 7, is the only one available when using a centralized version control system, but it can be adopted for a distributed version control system as well. A specific repository serves as a central rendezvous point to which developers contribute their changes. In a distributed version control system a developer pushes his modifications to that centralized repository and from there also pulls the changes of his colleagues.

It is challenging to setup this workflow for a distributed version control system, as it usually prohibits developers from pushing changes to the centralized repository. This setup combines not only the advantages, but as well the disadvantages of both approaches: Changes are immediately available, but if the centralized repository fails, the entire system fails.

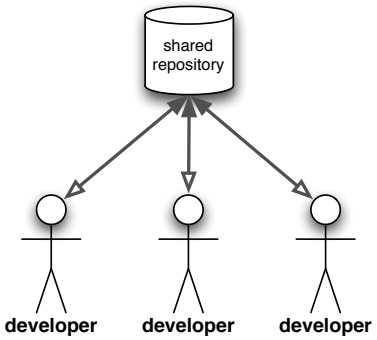


Figure 7: The centralized workflow, which is used for distributed as well as for centralized version control systems

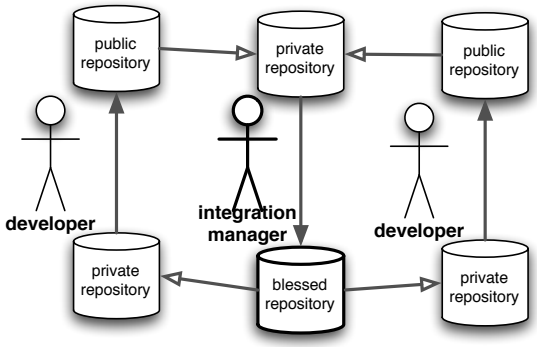


Figure 8: The integration manager collaboration workflow

Integration Manager

The **integration manager collaboration workflow**, drawn in Figure 8, aims to control all contributions to form a "blessed" repository, from which a clean version of the project can be obtained. Developer have their local repositories, which they update using the "blessed" repository. There is a second, public repository for each developer, where their contributions are pushed to. Inspired by the Rational Unified Process [Kru03] there is an integration manager, who is the only one who pulls changes from those public developer repositories. After combining and inspecting them the integration manager pushes them to the "blessed" repository, to which only she has (write) access.

Lieutenants

Figure 9 exemplifies the **lieutenants collaboration workflow**. Developers share their changes with a lieutenant. The lieutenants are each responsible for a subproject. They filter all contributions and decide, which of them to share with the dictator. The dictator integrates all changes into the "blessed" repository, to which only he has access. Updates are pulled from the "blessed" repository, which represents the latest version of the product.

This workflow is used for the development of the linux kernel, for which Git was developed. In this project Linus Torvalds has the role of the dictator. The lieutenants are specialists in specific areas (networking, video, etc), who are well known to the dictator. He trusts them almost blindly, as they handle the contributions of the community.

1 http://github.com

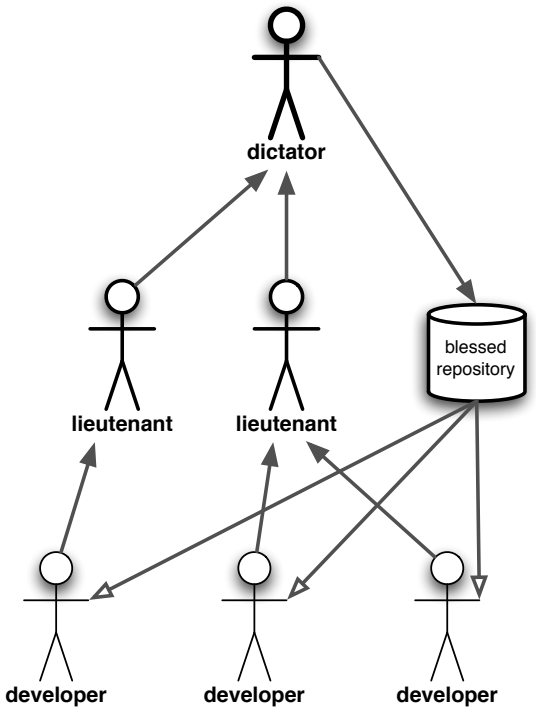


Figure 9: The lieutenants collaboration workflow

4.2 THE FREQUENCY OF COMMITTING CHANGES

There are two contradictory recommendations on how frequently to share changes. On one side, snapshots should be committed as frequently as possible to enable fine granular rollbacks. Committing changes late makes them more difficult to merge, as potentially more edited lines in the artifacts can be conflicting. An analysis with real world data in [EC95] came to the same conclusions. On the other side, shared changes should not corrupt the project (e.g. resulting in an uncompilable source code project), hindering further work of collaborators. Conflicts, occurring when two snapshots are based on the same version, can be resolved more easily, if the changes made are small, but occur more often due to the developers committing their snapshots frequently. Ideally, frequent commits should not be shared with other developers, if they leave the project in an unstable (e.g. uncompilable) state. After a task is completed all recorded changes should be shared.

This problem is evident in ***centralized version control systems***, as every committed snapshot is automatically shared with all participants. Working in branches could solve this problem but unless developers work in their own branch there are colleagues which are affected. Branches in most centralized version control systems do not come for free. In some projects build tools emphasize this problem. By running automated tests on the snapshot to be committed they prohibit fine-granular commits.

Distributed version control systems address this issue by differing between local commits and global pushes. Local commits are not shared with other users and can therefore be used to keep track of fine granular and possible destabilizing snapshots (e.g. that hinder a project to be compiled). A push shares all unshared local commits at once, but with only one participant. In practice there is a person, who collects all contributions and publishes them on a central place, as outlined in Section 4.1.

4.3 CONFIGURATION MANAGEMENT

Our solution focuses on version control only. The field of software configuration management, however, consists of four areas: (1) change management, (2) build management, (3) configuration and variant management and (4) version control.

1. Change management, i.e., tracking bug reports and feature requests, can be separated from version control. In fact supporting tools are often connected to version control tools. A common practice, however, is to mark the bug report/feature requests number in the comments of a commit. Some tools[2] automate this practice. As existing tools and practices can be reused with our prototype it was not necessary to design a decentralized change management solution as well.

2. Build management handles the build process of a project, which can be executed automatically, e.g., to produce nightly builds. These builds can be carried out on a local machine, thus existing solutions could be reused without difficulties. It depends on configuration management. Often tools[3] combine configuration and build management.

3. Configuration and variant management memorizes the possible configurations, which can be used to build a system, in different variants. It typically dissolves dependencies among different *item* versions as well. An item is not only a file, but can also be a tool, e.g., a compiler, which is necessary to build a project.

4. Version control finally records changes to files. It aims to reproduce the state of a project at an actively recorded point in time.

Our solution focuses on version control only, however, version control cannot be strictly separated from configuration management, as each retrieved state is in fact a configuration. Two mechanisms enable configuration management: We already saw that a complete underlying distributed version control system is reused, which includes its features for configuration management as well. These are snapshots, which are used to reconstruct a committed local state of any user. Each snapshot entry in the global history existed at one time in a user's local workspace. Therefore versions of files belonging to the same snapshot can be considered as depending on each other. The existence of named and unnamed branches is the other feature which enables variant management. Alternative versions of a project can be tracked in parallel using these branches. Lastly specific versions of a file can be selected from any snapshot, allowing the combination of any file versions in a local working copy (which is called cherry-picking). As no dependency information is provided a user must already know which files might fit.

4.4 CONSISTENCY AND COHERENCY IN VERSION CONTROL SYSTEMS

Consistency and coherency are the most important ***quality aspects*** of a version control system. They depend directly on a system's design.

Consistency and coherency describe the state in which the distributed copies of a system are. In general, multiple copies are consistent if certain integrity constraints are valid. These integrity constraints are to be defined for concrete areas and systems, in which consistency is analyzed. For database systems integrity constraints could demand certain statements to be true (e.g. an address entry refers to exactly one name entry). Most systems should never be in an inconsistent state, thus the defined integrity constraints can be understood as invariants.

The consistency definitions for caches and replicas, as defined in [HP98] , demands copies to be always identical. The weaker demand of coherency is fulfilled, if copies are updated on read access so that the read value is identical, regardless from which copy it was read.

2 like myLyn (http://www.eclipse.org/mylyn/)
3 like maven (http://maven.apache.org/)

The definition of consistency in the area of distributed shared memory (DSM) systems, outlined in [TS06], is quantified using the access time of concurrently operating users. Depending on when a read operation retrieves the latest written value, a certain degree of consistency is provided. Mosberger introduced in [Mos93] different **consistency models** for DSM systems. These consistency models can be understood as a contract that guarantees the retrievability of certain values when a distributed stored variable is updated concurrently.

The following scenario is usually used to explain the different consistency models: A process is writing a value to a variable, which is stored in different copies, among multiple locations. The implemented consistency model describes which values can be expected to be retrieved (the outdated or updated value) when subsequent processes read the variable from different locations concurrently. Multiple combinations are possible[4], each complying to a specified consistency model.

In these definitions of consistency updates replace the content of a value. In a version control system we create a new ***version*** of an ***artifact***, instead of overwriting its previous content. A specific version is immutable and never ***outdated***. Therefore we adapted these definitions to be applicable to version control systems where stored items are immutable and updates on their content create new items.

4.4.1 *Terminology*

In order to define the consistency and coherency for version control systems we need to define terms used in this area.

A ***version*** is a recorded state of an ***artifact***. When an artifact is created and modified multiple times, whenever a user executes the ***commit*** operation, the momentary state is saved, i.e., the content of the artifact is recorded and is retrievable later. The modern version control systems are capable of recording ***snapshots*** instead. A snapshot is created by recording the momentary state of all artifacts in a ***working copy***. In this way, not only the content of an artifact, but also the content of all other artifacts at the time the snapshot was created can be reconstructed. When we quantify the consistency of a system it does not matter if we investigate a single artifact's versions or the snapshots of all artifacts in a working copy. Both storage structures are equivalent regarding the provided consistency, therefore in our definitions the word *version* could be substituted with *snapshot*

$$\textbf{version or snapshot: } v_n \in V, \text{ where V represents all version/snapshots, for } n \in \mathbb{N}$$

denotes an arbitrary version of an artifact (or an arbitrary snapshot of multiple artifacts).

When the content of an artifact or working copy is changed and a subsequent ***commit*** records those changes, the resulting version v_n or snapshot is **based** on the changed version or snapshot v_{n-1}. We define this relation $\leftarrow$ as:

$$(v_{n-1} \leftarrow v_n) := \begin{cases} 1 & \text{if } v_n \text{ is based on } v_{n-1}, \quad \text{for } n \in \mathbb{N} \\ 0 & \text{otherwise} \qquad\qquad\qquad\quad \text{and } (v_{n-1}, v_n) \in V \end{cases}$$

The relation $\leftarrow$ is transitive.

Any two versions are called **related**, when they are on the same ***path***, i.e., being in the transitive relationship '$\leftarrow$'.

When the state of an artifact or working copy is recorded for the first time, an

$$\textbf{initial version: } v_{init} = \{v_n \mid \neg\exists v_x : (v_x \leftarrow v_n)\}, \quad \text{for } (init, n, x) \in \mathbb{N} \text{ and } (v_{init}, v_n, v_x) \in V$$

is created.

4 E.g. the first reading process could get the outdated value, the subsequent and every following process the updated value, which would comply to the sequential consistency model.

Multiple related versions, starting from an initial version v_{init}, form a **development line**. The last element in a development line is called the **head version**, if

$$\textbf{head}(v_n, t) := \begin{cases} 1 & \text{if } \neg\exists x : (v_n \leftarrow v_x) \text{ at time } t, \text{ for } (n, x) \in \mathbb{N} \\ 0 & \text{otherwise} \qquad\qquad\qquad\qquad \text{and } (v_n, v_x) \in V \end{cases}$$

Otherwise a version is **outdated**. N.B. The point in time is relevant to notice if a version is outdated.

More then one version can be based on the same version, formally described by

$$v_a \leftarrow v_x,$$
$$v_a \leftarrow v_y, \text{ for } (a, x, y) \in \mathbb{N} \text{ and } (v_a, v_x, v_y) \in V$$

The *development lines* ending with v_x and v_y are diverging from each other, which occurs when *branches* are stored in a version control system. Therefore we use the term *development line* interchangeable with the term *branch*.

For each development line a *head version* exists. We define the operation **heads**: $V, t \rightarrow V$ that gives us the set of all head versions that exist at a specific time t as

$$\textbf{heads}(t) = \{v_h \mid head(v_h, t)\}, \text{ for } h \in \mathbb{N} \text{ and } v_h \in V \text{ at time } t$$

A version can also be based on multiple versions, which represents *merged* development lines:

$$v_x \leftarrow v_a,$$
$$v_y \leftarrow v_a, \text{ for } (a, x, y) \in \mathbb{N} \text{ and } (v_a, v_x, v_y) \in V$$

If more than two development lines are merged, we call it an *octopus merge*. N.B. Multiple development lines form a directed, acyclic graph (DAG).

All versions which are (transitively) connected through the relation $\leftarrow$ are on the same **path**. The set of all versions in a given path includes all version of parallel branches as well, i.e., they are obtained by traversing all related versions from a given version to another given version. This is described by the following operation **path**:$V^2 \rightarrow V$

$$\begin{aligned} \textbf{path}(v_x, v_y) = V_{path} \Leftrightarrow V_{path} = \{&v_x, v_n, v_{n+1}, .., v_{m-1}, v_m, v_x \mid \\ &(v_x \leftarrow v_n) \wedge (v_n \leftarrow v_{n+1}) \wedge .. \wedge (v_{m-1} \leftarrow v_m) \wedge (v_m \leftarrow v_y)\}, \\ \text{with } x < n < m < y \text{ for } &(n, m, x, y) \in \mathbb{N} \text{ and } (v_n, v_{n+1}, v_{m-1}, v_m, v_x, v_y) \in V \end{aligned}$$

4.4.2 *Degrees of Consistency*

Updates on an artifact in a version control system create a new version (or snapshot) instead of overwriting the old stored value. To the author's best knowledge there is no definition of consistency and coherency, which takes this behavior into account. These definitions assume that a stored value gets replaced by an update. Therefore we modified the consistency models for distributed shared memory systems introduced in [Mos93] to be applicable to version control systems. To avoid confusion we call our introduced modified consistency models **degrees of consistency**. N.B. The introduced consistency degrees are not complete, but sufficient to describe the consistency of the related version control systems, which are presented in Chapter 5. There are certainly further degrees possible.

A certain **consistency degree** is provided by a system if it guarantees to retrieve a set of versions, even in the worst circumstances (with the exception of malicious attacks). However, if the circumstances hinder the retrieval operation to be executed successfully at all and not even a partial result is retrieved, the guaranteed consistency is not considered to be violated.

To define the consistency degrees, let us consider the following scenario: A user has an *outdated* version of an artifact or snapshot of a working copy, which he updates, executing

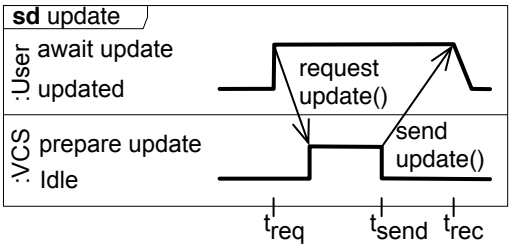


Figure 10: Timing diagram of the update process

an operation which retrieves the latest versions or snapshots of all ***branches*** of the respective artifact or working copy. For our definition it does not matter, if a user requests an update or an update is sent proactive by the system. Figure 10 shows the temporal order in this scenario. The user requests an update of a specified artifact or a working copy at the time t_{req} by executing the operation *request update()*. The version control system (named VCS in Figure 10) prepares the update and sends the user versions or snapshots that he does not store locally at time t_{send}. The user receives the resulting versions or snapshots at time t_{rec} and applies them.

Which versions or snapshots are retrievable from the version control system depends on the ***consistency degree*** it provides and is detailed below. We call the set of receivable versions or snapshots $V_{retrievable}$. Only, if the retrieve operation failed (or no version has been committed) this set is empty. N.B. The ***head version*** at time t_{send} can be outdated when the user receives it, at time t_{rec}, because a succeeding version could have been submitted by another user while the updating message is transferred to the user. $V_{retrievable}$ might only include heads(t_{send}) and not heads(t_{rec}).

The time difference between sharing a new version and retrieving it is considered indirectly by the consistency degree. It is mainly quantified by the ***freshness*** and changes depending on the circumstances a system experiences. In contrast to other ***quality aspects*** consistency and coherency do not change over time.

Like the consistency models our consistency degrees are sorted from loose to stricter. A stricter degree, like the ***sequential consistency***, narrows the possible results of the described scenario, but negatively influence on other aspects of a distributed system. A lose degree, like the ***eventual consistency***, offers more freedom to optimize other aspects of a distributed system, but tolerates undesired results. To achieve a strict consistency degree in a system multiple update messages to synchronize the distributed storages are necessary, wherefore the number of copies should be minimized. Less strict degrees require fewer coordination messages and allow more copies to exist.

Freshness

The **freshness** of an update is a ***quality aspect***, which measures how recent the retrieved results were pushed. Its ***metric*** is the time span between a successful ***push*** of one user and a subsequent ***pull***, which retrieves the just pushed changes.

eventual consistency : There are no guarantees when or which versions a user can retrieve. The retrieved versions can be completely ***unrelated***, with gaps in the retrieved ***development lines***. In the latter case a version is received while the predecessor version, it was ***based*** on, is missing. Formalized:

$$V_{retrievable} \subseteq \bigcup path(v_{init}, v_h \in heads(t_{send})), \quad \text{for } (h, init) \in \mathbb{N} \text{ and } v_{init} \in V$$

The only given guarantee is that after an unlimited amount of time all created versions are shared between all users. In typical implementations new versions have to be propagated to all replicas in the system. This usually occurs in an automatic process, in a deterministic, but coincidental fashion.

Eventual consistency is similar to the *eventual data consistency* listed by [TS06].

causal consistency : A system is *causal consistent*, if all ***related versions*** are retrieved. Unrelated versions might be missing. Gaps in the development line are avoided, which can happen in the weaker ***eventual consistency*** degree. Whenever a repository is split into disconnected parts, users with access to only one part can retrieve a different set of versions than users connected to another part. Additionally it might happen that a ***head version*** is only present in one part[5]. In a system which promises *causal consistency* all retrieved versions have to be ***related***. Formalized:

$$\begin{aligned} &\text{Let } H \subseteq \mathtt{heads}(t_{any}), \text{ where } t_{any} \leqslant t_{send} \\ &V_{retrievable} = \bigcup \mathtt{path}(v_{init}, v_h \in H), \text{ for } (h, init) \in \mathbb{N} \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{and } v_{init} \in V \end{aligned}$$

When having multiple ***head versions*** in multiple branches not all versions from all branches might be retrievable. The premise that given enough time all versions will be retrieved *eventually* holds true for causal consistency as well.

This consistency degree is similar to the *causal consistency model* defined in [HA90].

sequential consistency : In a system that guarantees sequential consistency, received updates include *all* versions in all branches, which existed when the update was sent to the user, i.e., at time t_{send}.

However, the ***freshness*** can have any value, and versions, which are committed after the update was sent are not received. Thus the latest received (head) versions might not be the latest existing versions. Formalized:

$$\begin{aligned} V_{retrievable} = \bigcup \mathtt{path}(v_{init}, v_h \in \mathtt{heads}(t_{send})), &\text{ for } (h, init) \in \mathbb{N} \\ &\text{and } v_{init} \in V \end{aligned}$$

Sequential consistency is typically guaranteed if the repository is controlled by a single machine, which serves requests in a sequential order. The transport time of a message is different for each pair of communicating machines, but very similar for subsequent messages between the same machines. Therefore it can happen that the latest versions shared by one user are not retrieved by another user.

This consistency degree is similar to the *sequential consistency model* introduced in [Lam79].

4.4.3 *Coherency in Version Control Systems*

Coherency describes the demand that identically named versions (or snapshots) are identical along all stored copies in a system. Having a relation $\mathtt{content}()$, which maps a version to its content, and a relation $\mathtt{identifier}()$, which maps a version to its identifier, a system is consistent if, and only if

$$\mathtt{identifier}(v_a) = \mathtt{identifier}(v_b) \quad \Rightarrow \quad \mathtt{content}(v_a) = \mathtt{content}(v_b), \text{ for } (v_a, v_b) \in V$$

5 N.B. An update based on a missing head version would reintroduce it to the system. Only, if a user based his changes on a previous version (as he might not have the head version as well), the head version would remain missing.

As long as a system is consistent, it is also coherent. An incoherent system would retrieve contradictory results to the read requests of different users. None of the existing version control systems can be in an incoherent state. In the description of selected version control systems in Chapter 5 this fact is justified.

4.5 SUMMARY

We have seen some of the common properties all version control systems share. There are different workflows which evolved while developers used the existing tools. The centralized solutions gather all created versions in a central place, following the ***centralized collaboration workflow***. The ***integration manager collaboration workflow*** or the ***lieutenants collaboration workflow*** filters the developers contributions by introducing integration managers in a hierarchical fashion. While a centralized version control system cannot implement the latter two workflows, a decentralized solution can be used with all three workflows. However, setting up the centralized workflow is not straight forward.

In regards to the summarized workflows, we discussed the impact of sharing changes with a lower or higher frequency. Submitting small changes in short distances allows a history with which changes can be reversed in a fine grained manner. Sharing only complete modifications, which present the final result of a task makes the system more user-friendly for other project members that want to have working updates only. We will see how the existing version control systems handle this trade-off between recording fine grained modifications for personal benefit and working changes only for the sake of coworkers.

As the most important ***quality aspects*** of version control systems we defined ***coherency*** and different levels of ***consistency***. We derived our definition from the area of distributed computing. Basically, the more the retrieved versions are connected, the higher the ***consistency degree*** a system guarantees is.

5 NOTABLE VERSION CONTROL SYSTEMS

The complexity of software projects increased rapidly following the development of the first programs in the 30s. By the late 60s, programs had become so complex that many projects failed. A solution to the so called "software crisis" [Som10] was version control systems, which helped to control the complexity of large projects and coordinated concurrent work on them.

In this chapter the most successful and thus influencing solutions are presented. After a brief introduction to these solutions their properties are discussed. In the conclusion of this chapter we will see why none of those solutions fully satisfies our requirements stated in Section 3.2.

5.1 CENTRALIZED VERSION CONTROL

The first version control systems can be classified as **centralized version control systems (cVCSs)**. These systems rely on a centralized control, as the client-server or mainframe communication paradigm does. This central instance handles all operations that are initiated by multiple users.

5.1.1 *SCCS*

The very first system that fulfills some of the requirements stated in Section 3.2 is called *Source Code Control System* (SCCS) [Roc75], which was developed in 1972. It was based on the **mainframe communication paradigm**, where a strong (and in those days expensive) machine executed all operations that users initiated through terminals. The terminals themselves did not have any computational capacities and served as input devices only. Nevertheless, SCCS can be installed on any machine - files are edited and stored as new versions on the machine that SCCS is running on. Some modern version control systems, such as BitKeeper[1], are based on SCCS.

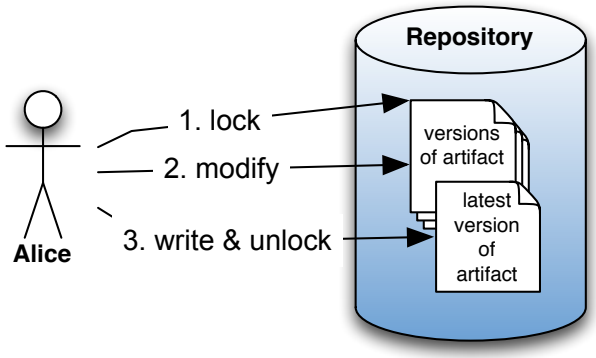


Figure 11: Basic architecture of SCCS and RCS

As depicted in Figure 11, all files reside on a central machine, in a repository. Users lock artifacts (so that no other users can make concurrent changes), modify and commit them, thereby creating a new version. The versions are stored using ***delta compression***, saving storage space, which was very expensive in the early days of computing.

1 http://www.bitkeeper.com/

This first solution solved requirement R-1.4 and requirement R-1.7 by controlling the modification of single artifacts. There was no support for ***snapshot*** version control (requirement R-1.5). The ability to work concurrently, however, was limited, as only one user could change an artifact exclusively. There was no support for requirement R-1.10: *Track variants (branches)* either. Versions of an artifact were stored in linear order only. Basic security needs were fulfilled with user account based requirement R-22: *Access and usage control*.

SCCS guarantees ***sequential consistency***. Following the ***lock-modify-write*** principle, an artifact is locked by a user who modifies it directly on the server. Contradictory changes are prevented, thus the repository, which exists as a single copy on the serving machine only, is always in a coherent state. The high consistency degree is payed by a limited ability to work concurrently, as one artifact can only be modified by one user at a time.

Pessimistic Concurrency Control

The **pessimistic concurrency control** locks an artifact, so that it can be read by anyone but modified by only one person. A technique, which is used by SCCS and RCS is called **lock-modify-write**, where an artifact is locked, modified and applied to a repository as a new version, which is unlocked. By avoiding concurrent versions to be created, **conflicts are resolved proactively**.

While preventing contradictory changes, this procedure obviously limits the possibility to work simultaneously on the same artifacts. When working on a software project, typically a group of artifacts have to be modified. This led to the behavior to lock more artifacts then necessary, which hindered concurrent work even more. The ***branches*** introduced in RCS relaxed this problem, as versions in different branches can be locked by different users.

5.1.2 *RCS*

To support variants (requirement R-1.10) the **revision control system RCS** (RCS) [Tic82] was developed in 1982, replacing SCCS. The system was based on the same basic architecture as SCCS, shown in Figure 11. The main difference is the ***version model***, which allows for the creation of a version, which exists parallel to another version, in a ***branch***. The main development line is called the **trunk**, where alternative development lines can split into parallel development lines, called **branches**. The system is additionally able to merge a branch back to the trunk - if the changes were not conflicting, i.e., did not happen in the same text line, the two versions (from the trunk and from the branch) could be merged automatically. This newly created version has two parents (the version in the trunk and in the branch it is based on) and marks the branch as merged back into the trunk.

Version Model

A **version model** defines *what* is controlled by a version control system and *how* it is done. It represents therefore the internal structure in which the version control information is stored. The *granularity* of control can cover single artifacts or entire projects, they can be tracked in revisions only or with variants, in a directed acyclic graph structure, where alternative ***development lines*** branch and merge again. They can be stored using ***delta compression***.

As in SCCS, ***sequential consistency*** is guaranteed, as the repository is stored and modified in the same way. Concurrent work is therefore limited by the ***lock-modify-write*** approach, although the introduced branches allow for limited parallel work [KR81].

5.1.3 *CVS*

RCS was eventually replaced by the **concurrent versions system (CVS)**, which began as a collection of scripts operating on RCS in mid 1984. Back then, it evolved from the need to coordinate the work of three developers, who each had different working times, and was

called *cmt*. CVS was released in late 1990s and is still used in many projects. It does not rely on RCS anymore, as it became an independent software project.

Optimistic Concurrency Control

The **optimistic concurrency control** allows users to make simultaneous changes to artifacts. As only a small number of concurrent changes are conflicting, as discovered in [Leb95], the advanced version control systems, which were developed after RCS, implement the **copy-modify-merge** procedure described by [SS05]. A user copies an artifact in a specific version from the repository, modifies it, and ***commits*** it back to the repository, where a new version is created. If a new version already exists, the user must ***merge*** the contents of his version and the latest version of the modified artifact and commit the resulting version. This process is called **reactive conflict resolution**.

The ***distributed version control systems*** are unable to detect a conflicting version when it is created; they therefore implement the **copy-modify-branch** procedure. If multiple versions are based on a common base version they reside in ***unnamed branches***, which should be merged, creating a new version based on both merged versions.

The architecture is still centralized, although it moved from the mainframe to the **client-server communication paradigm**, where a powerful machine serves the requests of multiple client machines, which each have enough power to make calculations themselves, in contrast to the terminal machines. Nevertheless all operations are executed on the server machine.

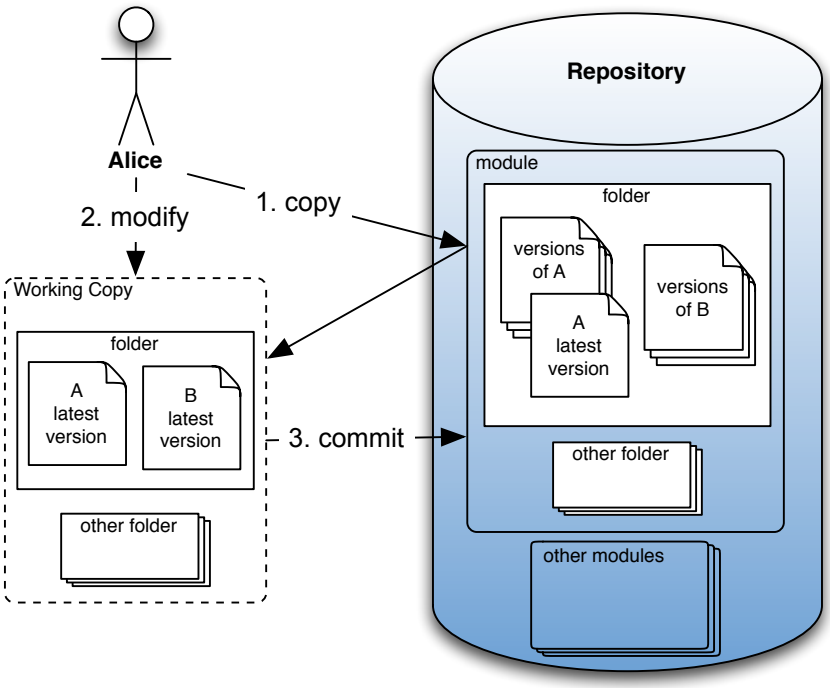


Figure 12: Basic architecture of CVS and Subversion

Initially CVS eased the work on a set of artifacts, as RCS and SCCS merely controlled the versions of a single artifact. CVS raised this scope to a project level, where all artifacts of a project are controlled. Figure 12 shows the evolved basic architecture. Artifacts are grouped into folders, which may include subfolders with further artifacts. The outermost folder encloses the software project and is tracked in a **module** on a CVS server. One or more instances of those modules form a **repository**. A software project can be organized in multiple modules,

although no connection exists between the modules in the CVS. A user can **check out** a module, which creates a *working copy* on his machine.

Working Copy

A surveyed folder is called the **working copy** of a module. If anything in this folder changes, the version control system captures all of the modifications. With the ***commit*** operation, a new ***snapshot*** is taken locally. At any time the folder (or individual files) can be reverted to a formerly recorded state. All snapshots are ordered, forming a history. When the user changes several files and executes the commit operation, a new snapshot is created, which records the latest snapshot as being its parent.

This *working copy* mirrors the artifacts and folders of the respective module. Initially all artifacts are *copies* of their latest version in the server's repository. A user can now make any changes to the artifacts and *commit* his changes back to the module stored on the repository, which creates a new version of each changed artifact. New folders and new artifacts in the working copy are also created in the module on the server as a result of the commit operation. However, renames are not supported, instead the old artifact/folder is deleted and create as a new one (with a changed name, and no former history).

Modified artifacts can only be committed when they are based on the latest version on the server. If another developer committed his changes earlier, these changes must first be retrieved and then merged with the ones modifications to be able to commit all local changes. All operations affect the entire module, however, the versions are still tracked for each file individually. Thus it is not possible to map the versions of different artifacts to each other, e.g., *checking out* an artifact in a specific version, which was *committed* at the time another artifact's version was checked in. The state of single files is tracked, but not the state of the working copy as a whole. Alternative *development lines*, branches, are also supported.

The introduction of *working copies* enable an *optimistic concurrency control*. Files can still be locked, but following the *copy-modify-merge* approach enables users to work at the same time on the same artifacts.

The price payed is the weaker *degree of consistency* in comparison to the *mainframe communication paradigm* based systems: *sequential consistency*.

Coherency is never compromized. Contradictory changes, which lead to version conflicts, cannot be committed. Solving them forms a new version, which can be committed.

5.1.4 *SVN*

Subversion (SVN) was developed to overcome the limitation of CVS, tracking the *history* of single files only, by the company CollabNet [Col], that originally developed CVS. It was released in late 2000. Although adopted by a huge number of projects, it never replaced CVS completely. Subversion shares its basic architecture with CVS, which is presented in Figure 12. The terminology, however, was altered slightly: The equivalent of a working copy is a repository, multiple repositories reside in a database on the server. A repository is not used exactly like a module in CVS. Usually the outermost folder of a project (or a working copy) is inside a folder in the repository. By convention, a Subversion repository has the following three folders as its outermost folders: trunk, branches, and tags. Initially all files of a project are stored inside the trunk folder.

When branches are to be created, a *shallow copy* of the project is stored in the branches folder (in a subfolder named like the created branch). A project can be tagged in the same way, by storing it in the tags folder. From any folder level the *update* operation can be invoked, which copies a specified version of all artifacts and folder encapsulated by this folder into the working copy. The commit operation, however, creates one version for all artifacts and folders in the repository. Thus the state of a working copy at a specific time is recorded, rather than the state of single artifacts/folders, like in CVS. When a specific version is retrieved, all artifacts and folders are in the state in which they were committed.

Shallow Copy

A shallow copy is a concept, whereby data is copied without using additional storage space. Stored data is addressed by an identifier. A shallow copy creates a new identifier, which refers to the data's storage address instead of creating new storage entries and referring to them, as would be the case in a classical deep copy. This concept has been applied by different mechanisms. A link in a file system to the actual content and a pointer in a programming language to the actual data are two examples of a shallow copy.

Shallow copies are created faster than normal copies, and do not consume additional storage space. A drawback is, however, that modified data is modified for all identifiers referring to that data. A mixture of a deep copy and a shallow copy is a **lazy copy**, where data is initially copied as a shallow copy. Upon the first write access, the modified data is stored under a new address instead of being overwritten in the previous place. Only the shallow copied identifier refers to that new storage address, as it would have done if it had been copied as a deep copy.

Unchanged artifacts in stored snapshots in Subversion are stored as shallow copies (in a database), while cloned repositories in Mercurial [Mac], 5.2.4 are copied as lazy copies (in a filesystem; using the command 'ln' in unix based operating systems, 'mklink /H' in Windows). As soon as an artifact is changed the created new version is stored in a new database entry in Subversion and automatically in a new storage location on the hard disc when using Mercurial. In both cases the new version now requires additional space.

However, the state of a user's working copy is not tracked correctly. Let us assume that we have a repository, only consisting of two artifacts: a and b. Initially two users, Alice and Bob, have identical copies of a and b in their working copy. Alice only changes the artifact a, and Bob only changes the artifact b. If they never update their working copies (which is not necessary, as their modified artifact is always based on the latest version in the repository), and commit their changes concurrently, the resulting state of the repository includes both modified artifacts, although in each working copy as the other's artifact remains unmodified. Experiments showed this result, which can be explained by the fact that Subversion only tracks the modifications, not which other files were not modified. While requirement *R-1.2: Detect concurrent modifications* is fulfilled, this example reveals that requirement *R-1.6: Retrieve a recorded state* remains unfulfilled.

Requirement *R-8: Offline version control* is hard to fulfill using a ***centralized version control system***. Recently, the operation to compute the changes to artifacts in the ***working copy*** from the snapshot on which they are based (***diff***) and to *revert* those artifacts has been implemented, by caching the base snapshot on the client machine.

As explained before, the creation of branches is supported. In practice, merging branches is a difficult task, which was one factor that motivated the development of the distributed version control systems.

Subversion preserves ***coherency*** with the same mechanisms as CVS does, using the same ***optimistic concurrency control***. It therefore also guarantees ***sequential consistency***. In direct comparison with CVS, however, the consistency degree is higher: versions of artifacts can be mapped to the versions of other artifacts, which were committed at the same time (using ***snapshots***).

5.1.5 *ClearCase*

ClearCase represents a very different approach to the presented version control systems; Mainly because it supports configuration management, which is beyond the scope of this work. ClearCase is a commercial system offered by *IBM*, presented in [Leb95, BM05]. ClearCase was developed as the successor of DESS (Domain Software Engineering Environment) by the company *Atria Software* in 1992. This company merged with *Pure Software* and bought by

Rational Software, which was acquired by *IBM* in 2003. ClearCase is part of a tool environment which supports a complete software product development cycle, from project management to configuration management. The tool suite is called *Unified Change Management (UCM)*. The version control is handled by ClearCase in this environment ([JG03]).

The ***version model*** of ClearCase stores metadata, which is needed by configuration management as well. The stored metadata includes *version identifiers*, *tags* and *types*, which are used by the version control, as well as *attributes*, *hyperlinks* and *triggers*. Each versioned item has an *identifier*, which is a composition of a global sequence number and a local sequence number. The *type* of the versioned item is recorded to decide which tools can be used to compress or edit it. Therefore, not only text files, but proprietary files can be handled efficiently. *Attributes* are evaluated to derive a configuration, called *view* in ClearCase, which composes a working copy of selected artifacts' versions. *Hyperlinks* can connect items. E.g., if versions are merged a hyperlink connects them. This information can be used for traceability, satisfying requirement *R-10: Traceability*. *Triggers* define which action should be automatically executed if a specified event occurs. All version controlled items are stored in a **versioned object base** (**VOB**), which is ClearCase's repository.

A **versioned item** is an artifact, a folder, but a derived object as well. A **derived object** is the result of a build process, which transforms multiple artifacts into one product. When a versioned item is modified, a new version is created and stored. ClearCase uses configurations instead of snapshots to retrieve matching versions of different items. The system used is completely transparent to a user. Users work with the files, which appear as if they would be in their local file systems. If an operation modifies the content of an artifact a version is recorded automatically, which forms a fine-grained history. If a user has finished his task he can call an operation of ClearCase, which forms a *baseline*. This signals a stable (compilable, working) configuration. The latest versions of all items are automatically labelled to indicate that they belong to this baseline. This solves the problem described in Section 4.2.

A user works on a view, which is similar to the concept of ***working copies***. A configuration, usually provided by an integration manager, defines which versions of which version items a user works with. They can belong to any *branch*, so that items of different branches can be mixed. Items can be locked to prevent modifying them.

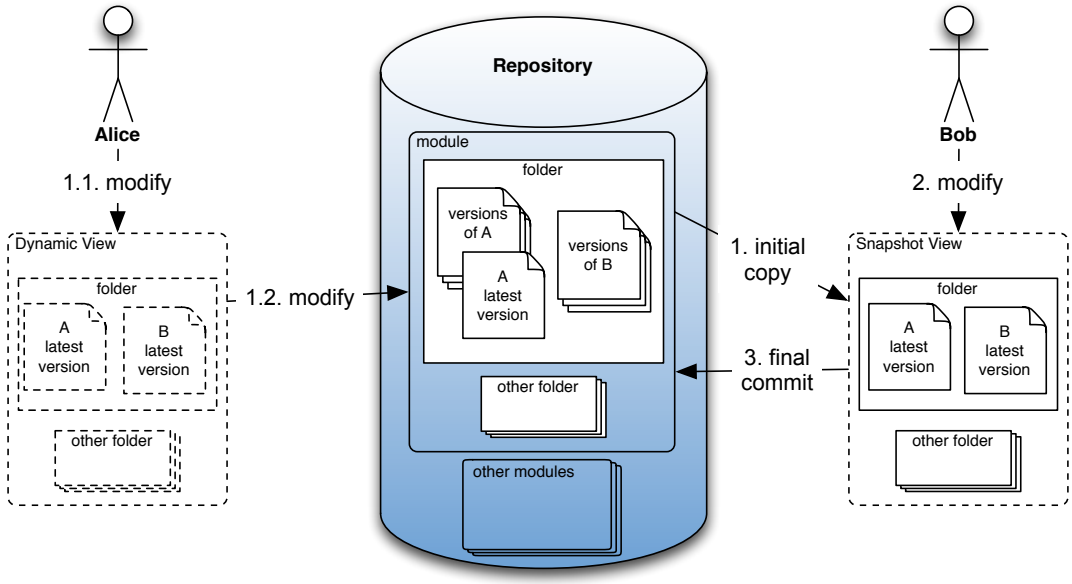


Figure 13: Dynamic (Alice) and Snapshot (Bob) View in ClearCase

A user can base his working copy on a *snapshot view* or a *dynamic view* as presented in Figure 13. In a *dynamic view* all files remain on a server. In the figure all files appear as being

on Alice's local local hard disc, but they reside in fact on the repository storing server. All operations, which change an artifact's content, are sent and executed on this server, which creates a new version. Changes are thereby immediately visible to all other developers. The time required to finish an operation depends on a user's network connectivity and is greater than if the operations were conducted on local files. A *snapshot view* is a copy in the user's local file system of the versions on the server. An administrator initially copies the required artifacts to Bob's local machine. Bob modifies these artifacts, where each operation results in an automatically created version, which resides on Bob's machine. If the new created versions should be shared, Bob or a distinguished administrator commits them to the repository on the server. Conflicts with versions submitted by other users are detected and have to be solved in this step. All local versions are applied in an atomic transaction to the servers repository.

In the *dynamic view*, no conflicts can occur, as changes are immediately visible to all users. If an editing tool does not notice that an open file has been modified, however, it will overwrite those changes. Though artifacts are usually locked to be modifiable by a single user only. Conflicting changes might occur in the *snapshot view*, which are resolved when a user intends to commit his local versions to the repository on the server. This is usually done through a distinguished administrator.

The setup presented in Figure 13 is possible but unlikely. Users can work on the same artifacts in the same branches, using different views. But as mentioned before, conflicting changes are to be avoided by locking files to be modifiable by only one user, or by allowing the development in different branches. A mixed setup is possible, where the artifacts, upon which multiple users work, exist in multiple branches, and other artifacts are locked to be modifiable exclusively by one user, where changes are immediately visible for all other users. It is also possible that, e.g., Alice has an exclusive lock on artifact A and Bob has one on Artifact B, while both see changes of the other's artifact immediately.

ClearCase makes the version control very transparent to the user. A user modifies artifacts without explicitly committing changes. ClearCase records all modifications in a fine grained manner. Only a distinguished user manages the repository, by assigning which versions are visible to which other users and which of those versions they are allowed to modify. Conflicts are usually prevented by giving only one user write access to an artifact in a branch. If multiple users are allowed to make concurrent changes, resulting conflicts are solved by the administrator.

There is an extension, called **multisite**, which aims to solve the low performance of a *dynamic view*, and fulfills requirement *R-18: Adequate communication speed*. In this setup, multiple servers are deployed. They are equal with respect to their responsibilities. Each server has a full copy of the repository. The content is, however, not always up to date. It is legitimate for the latest versions of some items or complete branches to be missing. These servers can be synchronized at any time, which can be initiated manually or be executed automatically; periodically or if a specified event takes place. To avoid consistency problems, each server manages a disjunctive set of versioned items and has an exclusive write access to these items. New versions can be only applied to the respective server's repository. In a global development setup a server would reside on each project team's location. Each developer can access each of those servers, with either a *dynamic* or *snapshot view*.

Atypical for a modern version control system, ClearCase implements ***pessimistic concurrency control***. An administrator configures which artifacts in which branches are allowed to be written by a user and locks these artifacts to this user only. All other users can read these artifacts and see updates, but they cannot change the content.

Coherency is always preserved by forbidding concurrent changes to the same artifact in the same branch. If conflicts occur, e.g., while merging branches, they are noticed by ClearCase and are to be resolved.

The guaranteed ***degree of consistency*** in ClearCase depends on the view a user works on. In the dynamic view ***sequential consistency*** is guaranteed, as artifacts are changed directly on a server. The snapshot view can offer more performance, as artifacts are modified on the local machine and transferred to the server at a later point in time. Thus only guaranteeing

sequential consistency. When using the multisite setup, the guaranteed consistency degree drops to ***causal consistency***. The latest versions of some artifacts might be missing on some servers. The users always try to update their versions from the servers, which store the latest versions. Nevertheless, some machines might not be able to provide any versions.

The transparent version control offered to a developer is only possible by giving an administrator the complex task to configure the system and to avoid conflicting changes by pessimistic locking or detect and solve occurring conflicts. ClearCase achieves this high degree of consistency with the costs of performance. All operations on artifacts in a dynamic view are delayed by the time the network messages need to travel to the repository storing server. Practically one can work in a dynamic view only if the connection is as fast as in a local area network (LAN). The dynamic view is not usable in a global software engineering scenario, even when a multisite server setup is used. The snapshot view works as well as if a Subversion server had been used, although conflicting changes are more likely to happen due to the postponed synchronization with the server.

5.2 DISTRIBUTED VERSION CONTROL

To overcome the shortcomings of centralized systems, mentioned in Section 2.4.1, a new approach arose with Monotone [mon] in mid 2003. Several **distributed version control systems (dVCSs)** followed, which differ only in a few details. In fact they behave, from a functional point of view, almost identically - especially Git [HT] and Mercurial [Mac] are very similar, taking their various extensions into account. They only differ in how they perform in specific tasks. Because of the strong similarity, the basic concept is shown first. A description of the most important dVCSs, Monotone, Git and Mercurial, follows. The dVCSs are popular among a great variety of open source projects. Only a few commercial projects exist, which prefer a dVCS to a cVCS. The reason for this might be the lack of a **global repository**, where the changes made by all developers are gathered immediately. However, for most commercial projects it is unknown which VCS is in use. Nevertheless, the first mature dVCSs (Git) was developed solely to support open source projects (to be more precise: one open source project: the linux kernel). As detailed in Section 4.1 not all developers' contributions are integrated into the open source project. A dVCS supports different repositories among its users, where only parts of a history is integrated into the history of another developer's repository.

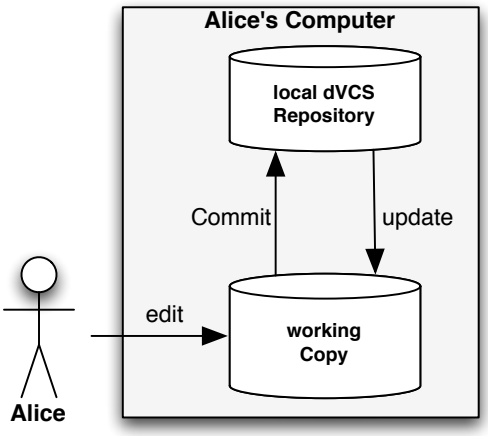


Figure 14: Basic architecture of dVCSs

5.2.1 *Basic Architecture of dVCS*

Common to all dVCSs is the distributed repository, shown in Figure 14. Instead of a centralized repository, every user has a complete repository on his local machine. The user (Alice) edits

artifacts in a ***working copy***. If she wants to store any changes made, she executes the *commit* operation, which creates a new snapshot in the associated local repository. Stored versions can be retrieved by issuing the *check out operation*. Note that here the term *repository* does not have the same meaning as in CVS, where a repository is the place on a server, where all modules are stored. As in Subversion a repository of a dVCS stores only one project with all metadata, such as branches, tags and snapshots.

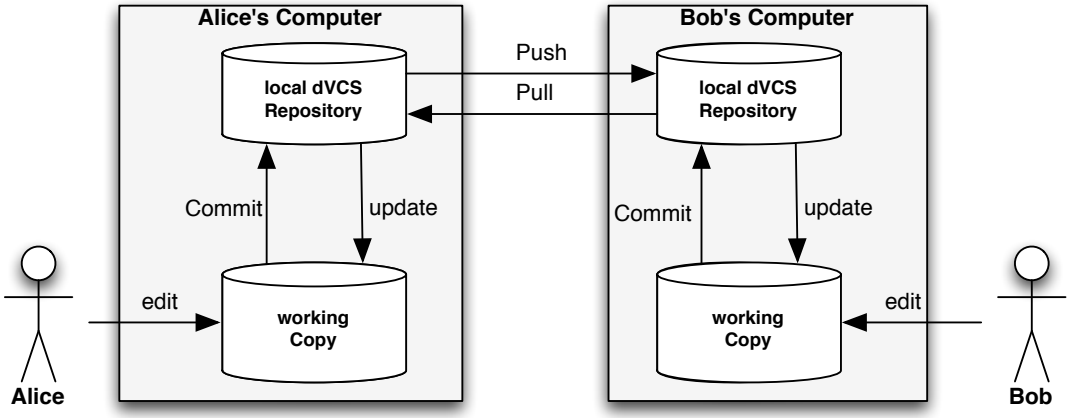


Figure 15: Sharing versions between participants of a dVCS

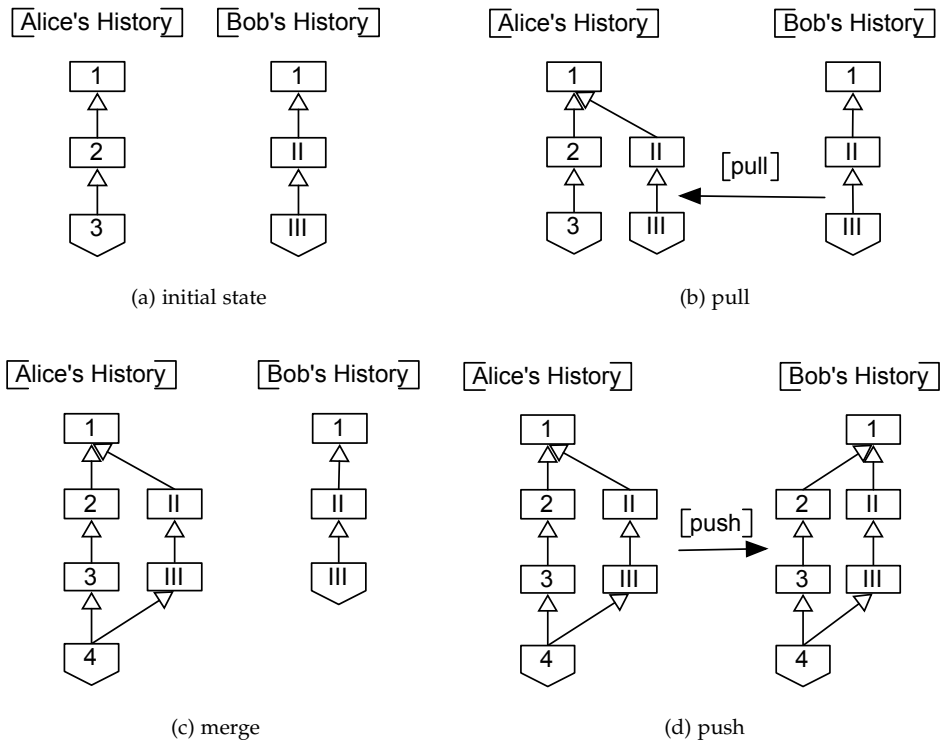
Figure 15 shows that two users share their snapshots. As each user has a repository, there is no main repository. Both, Alice and Bob, edit files on their computers' working copy and execute the commit operation to create a history of their changes in the local repository on their machines. To exchange each others history the *push* or *pull* operation is used, as demonstrated in Figure 16.

Alice and Bob created the history presented in this figure. The history is in the respective local repositories, as presented in Figure 16a. Alice and Bob started working with the same initial snapshot, named *1* and being the first snapshot in both histories. Alice and Bob both committed their changes to the artifacts in their working copy twice, creating the two subsequent snapshot entries in their history, named *2* and *3* in Alice's and *II* and *III* in Bob's history. Alice informs Bob (by e-mail, a phone call, etc.) that it is time to share their changes. In this process not only are the latest versions exchanged, but the complete history of both repositories is synchronized.

Alice pulls Bob's changes into her local repository (compare with Figure 16b). Bob has to execute the **serve** operation, which prepares Bob's repository to look for incoming connections. The two machines connect directly when Alice invokes the **pull** operation with the repository on Bob's computer as the target. A protocol specific to this dVCS transfers Bob's snapshots in this way to Alice's machine. Bob's snapshots are based on snapshot *1*, but might conflict with Alice's snapshots. Although they may be able to merge automatically, Bob's snapshots are stored as an alternative ***development line*** in Alice's history, called an **unnamed branch**. There are now two head snapshots in Alice's repository depicted in Figure 16b.

Alice can continue to work or merge the two head snapshots, *3* and *III* in her working copy. A subsequent *commit* creates snapshot *4*, shown in Figure 16c. Now she can push her history to Bob, which updates his history with the missing snapshots, using the **push** operation (see Figure 16d). N.B. Bob could have executed the *pull* operation to aquire Alice's snapshots as well - but Alice would have to enable the server mode in her dVCS first. If Bob continued to work, and committed further snapshots in the meantime, the new snapshots would still be in Bob's repository after Alice's push.

The previous example showed that there is no repository where all latest snapshots are stored, like a centralized repository in cVCSs. Each developer has his personal repository, in which he can integrate changes from other developers.



(a) initial state (b) pull

(c) merge (d) push

Figure 16: Alice and Bob share changes in a dVCS

Having a local repository creates a possibility, which is not present when dealing with a centralized repository: The history of the committed snapshots can be rewritten. That means that unwanted commits can be erased and snapshots can be reordered, to make the history look nice. A common use case is to develop in a branch. Instead of merging the branch into the main ***development line*** after the work is done the branch gets **rebased**. This operation puts all snapshots from the branch on top of the latest revision of the main development line, so that the version ***history*** looks like they where committed later. However, this has to be done before snapshots are exchanged, as the rewritten snapshots change their identifiers as well.

The absence of a centralized repository could be countered with a dedicated repository, which serves as a rendezvous point, to which all users synchronize their repositories. The snapshots stored in this repository, however, would be ***outdated***, as new snapshots are created locally and shared subsequently. This is one of many possible workflows. A further discussion of typical workflows can be found in Section 4.1.

The local repository offers all of the features of a centralized repository, including but not limited to (named) branches, tags, snapshot commits and updates, fulfilling requirement *R-1: Support common version control operations*. Network communication is only necessary for exchanging snapshots, as all other functions are executed locally. These functions operate, therefore, much faster than in centralized version control systems. ***Checking out*** a project initially, called **cloning**, however, is slower than in a cVCS, as not only the latest snapshot, but the complete repository has to be transferred to the developer's machine. To speed up this initial copy of the repository, it can also be transferred by, for example, an USB flash memory drive.

Due to the distributed repositories, artifacts cannot be locked by a developer, thus ***pessimistic concurrency control*** used with the first version control systems cannot be applied. The optimistic concurrency control used by most cVCSs, ***copy-modify-merge***, cannot be applied either: The copy-modify-merge procedure requires a communication with all repositories, which is not possible in a dVCS. Additionally, in this procedure a snapshot cannot be committed if it is based on the same base snapshot as an already committed snapshot. This

would be impractical in a dVCS, as the push/pull operation applies multiple snapshots to a repository. As stated before, any snapshot can be applied, forming an unnamed branch, if based on the same base snapshot like an earlier committed snapshot. The workflow used here is, therefore, ***copy-modify-branch***. All of the distributed repositories are always ***coherent***. Contradictory changes of the same artifact are saved in unnamed branches and the version identifier is calculated based on a version's content. As we saw in the previous example, unnamed branches can be created in a named branch. The development is no longer linear, but forms a direct acyclic graph (DAG).

The flexibility to work without a network connection comes with the price of guaranteeing ***causal consistency*** only. All snapshots in a user's repository are related to each other (except the most recent ones). But no user has all existing snapshots. The received snapshots depend on the user, who was manually chosen to ***pull*** changes from. Conflicts are only detected, when users exchange their snapshots.

5.2.2 *Monotone*

Monotone pioneered this new way to control the development of software projects in mid 2003 (see [mon]). The complete repository with all metadata is stored in a local database, which is saved in a file, using SQLite[2]. The metadata structure is very similar to the structure created by Mercurial, which is described in more detail in Section 5.2.4. There can only be one working copy per repository. A repository can be ***cloned*** locally by simply (deep) copying the database file, which consumes additional hard disk space. A second local working copy can then be created, e.g., to track an experimental development in a branch.

Snapshots can be exchanged between local or remote repositories, as demonstrated in Section 5.2.1. Monotone knows a third operation in addition to ***push*** and ***pull***: **sync**, which executes *push* and *pull* directly after each other. The only way to access a remote repository is with the custom protocol *netSync*, which is similar to *rsync*[3]. This protocol transfers the missing snapshots and other missing metadata only. The same protocol also transfers changes between local repositories, but to create a local repository, the entire repository has to be copied.

Monotone features a built-in security concept. Each developer has a public/private key pair, which is used to sign snapshots and grant access to ones repository. The access can be read only or read and write, but is set for the entire repository, not only individual artifacts.

Monotone is used by a small number of open source projects[4].

5.2.3 *Git*

Forking the development of software projects, by using branches, is often a difficult task. An important aspect of Git is a new approach to handle branches. Linus Torvalds initiated the development of Git in April 2005 (see [Tor]). Today it is the most popular ***distributed version control system***, used by many projects[5]. The Monotone version control system could have been used rather than developing a new system, as the possibility to work offline was compelling. However, it performed too poorly, although it was later improved by revision 0.27.

Creating and merging branches in Git was designed to be as easy as possible. In the philosophy of Git, every user's local working copy is a branch for another user. Thus, there is only one head version for each user, called master in Git, which is similar to the concept of a ***trunk*** in cVCSs. The master branch of one user is a normal branch for another user, who has his own master branch.

2 http://www.sqlite.org
3 http://rsync.samba.org
4 like coLinux (http://www.colinux.org), CTWM (http://ctwm.free.lp.se), Pidgin (http://www.pidgin.im) and Xaraya (http://www.xaraya.com), to name some
5 among them are the linux kernel, many linux distributions, including Android, Debian and Fedora, many GNU projects, including gcc and many other popular projects, such as Pearl or VLC, all listed in http://git.wiki.kernel.org/index.php/GitProjects

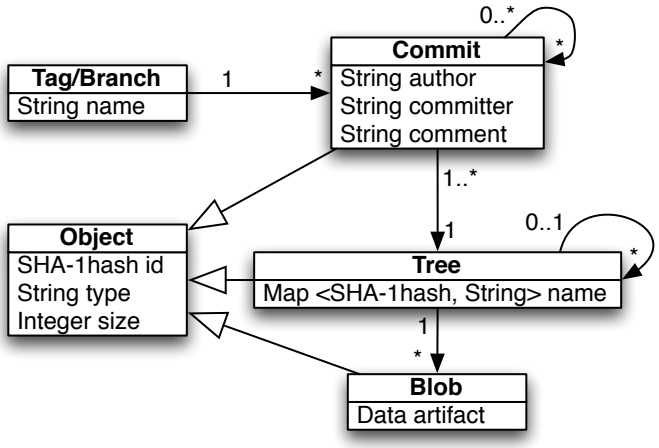


Figure 17: Metamodel of Git's version model

Figure 17 shows the ***version model*** of Git. There are *objects* and *tags/branches*. All these objects are implemented using text files, which are stored compressed. An *object* contains its type, which can be the string "blob", "tree" or "commit", its size on the hard disk and content, which depends on the type. The identifier of an *object* is computed by hashing the *object* with the *SHA-1* ***hash function***. A *blob* represents an artifact, the type is, therefore, the string "blob". It stores the content of the artifact it represents in textual or binary form. The name of the artifact is stored in a *tree*. A *tree* represents a folder. The content of a *tree* is a map of hash ID and string pairs, where the hash ID is either the ID of a *blob* (representing an artifact) or of a *tree* (representing a folder) and the string is the artifact's name or rather the folder's name. By listing all *blobs* and *trees* in a *tree*, the structure of the artifacts, folder and subfolder in the controlled working copy is mirrored. There is only one *tree* at the top level of this hierarchy, which represents the root folder of the working copy.

Hash Function

A **hash function** (often ***SHA-1***) calculates a **hash value** from a given input string. The resulting hash value is element of a limited codomain and has a fixed number of digits. It can happen that two input strings result in the same hash value, which is called a **hash collision**.

However, how likely a hash collision occurs depends on the used hash algorithm. If all possible input strings are known a *perfect hash algorithm* can be used where collisions do not occur at all. **Cryptographic hash functions** have a strong collision resistance to hinder the decryption of encrypted data. **SHA-1** (secure hash algorithm 1) is a cryptographic hash function, which was developed by the national security agency (NSA). Theoretically a hash collision can occur when using SHA-1, but the odds are negligible small. SHA-1 is utilized in the widely used *SSL encryption*, and to the date of this writing not even one accidental collision was reported.

A *commit* represents a committed ***snapshot***. It stores metadata about the commit: the author's name, the name of the committer and a comment. It additionally stores the hash ID of the topmost *tree*, which represents the topmost folder in the working copy. A list of hash IDs are included, which represent parent commits. In a linear development there is only one parent, but if the commit is the result of a merge, multiple parents are listed. That could be any number, because more than two branches can be merged in Git (called an **octopus merge**).

Identifying these objects with the value, which is computed by calculating the hash value of the object itself, and referring to each other in the outlined structure, has some positive side effects. The folder structure automatically refers to unchanged and changed artifacts, effectively creating a snapshot. Unchanged artifacts are not stored twice, the *tree* is referring

to the old artifact instead. If an artifact is changed a new *blob* is created, which stores the complete content of this new artifact, not only the changes to the previous version as it is usually done using ***delta compression***. The built-in compression, however, ensures that no storage space is wasted and is in most cases more efficient. Note that a changed artifact in a subdirectory not only creates a new *blob*, but for all folders on the path to that artifact a new *tree* is created as well. The new blob has another hash value, as the represented artifact's content changed. This new value is entered under the old name in the *tree* which represents the folder that contains this artifact. This changed content results in a new *tree* object, which has another hash value, which updates the *trees* which refer to it, etc.

In the same manner the history is cryptographically saved. The changes made to the artifacts and folders lead to a unique hash ID of the topmost *tree*, which is entered in the *commit* object. The hash value of that *commit* object is entered in the succeeding *commit* object, etc, building a self signing history. If one object would be manipulated the computed hash values of the other objects would reveal the manipulation.

Tags and branches are represented by small files, which only contain the 40bit hash value of a *commit* (and a 1bit newline). The filename of this *tag/branch* object is the tag/branch name. If a new *commit* is the successor of a *commit* a branch refers to, the branch is updated with the latest *commit's* hash id.

Regarding this structure, branches are extremely economical. In fact only 41 bits of space is required for a branch. Git encourages working with multiple branches. For each feature, a new branch should be created, and when working on different features at the same time, a developer should alternate between branches. E.g., Alice is working on a new feature, when her boss wants a bug in the code fixed. She commits her last changes to her feature branch and creates a new branch from the snapshot where her current work is based on. Now she updates the artifacts in her working copy to that commit and begins to work on the bug fix. She can switch quickly between the branches, as only the artifacts in the working copy have to be reverted. Only conflicting changes, which are recognized by having multiple *blobs* with the same name, need to be reconciled when merging the changes back. A common practice is to ***rebase*** the commits so that the history appears more linear, before sharing the changes with other developer.

The previous example showed how Git actually tracks the content of artifacts instead of the artifacts themselves. This has a positive effect on renamed or copied artifacts. As long as the content is identical, they are represented by the same *blob*. Their name is stored by the *tree* that represents the surrounding folder. Git detects renamed artifacts, which have had their content changed, using a heuristic, which probes the similarity of different candidates.

This slim storage structure, the ability to rebase the history and to share only selected branches is an improvement to Monotone. The ***history*** is exchanged using the ***push*** and ***pull*** operations. Just as Monotone Git offers a proprietary protocol, but this protocol can operate on top of http or ssh. Additionally, a patch, which includes all changes between specified snapshots, can be created locally and shared as a file in any way, including via e-mail or using an USB flash memory drive.

Git is able to synchronize with other version control systems such as cvs, Subversion or Mercurial. Depending on the concrete solution, this can be a (one or) two way synchronization or a continuous exchange, making the systems interoperable. Git could even be used to combine two other systems, working as an intermediary. There is also a cvs server integrated into Git, so that the repository can be manipulated as though cvs where being used. Adapters for other version control systems, such as Mercurial to name one, exists as well or are created by the community as extensions. The development of Git is open source and it has a good extension system, whereby optional functions can be hooked in.

As a drawback of the repository storage structure, it takes a while to traverse the history of a single file. It needs to be computed on demand by traversing the *tree* and *blob* structure of every *commit*. An operation dealing with this task is provided, although its execution is not recommended. Similarly, a change set, i.e., a list of the artifacts that were changed in a given snapshot (or commit in Git's terminology), must also be computed by comparing a

snapshot with its parent snapshots. A further drawback is the platform dependence of Git. It runs native on unix/linux based machines, including apples OS X. But it is difficult run it on a windows based machine. In essence, the unix compatibility layer cygwin[6] must be installed.

5.2.4 *Mercurial*

Mercurial shares almost all features with Git. Like Git, Mercurial is able to work in symbioses with other version control systems[7]. It could be used in a mixed setup, where different clients use different VCSs. Meanwhile many features were added using extensions; many of those are delivered with the basis package of Mercurial and are easy to activate. The ability to ***rebase*** snapshots, for example, was not planed initially (as the philosophy of Git was not shared), but was later added through an optional extension. Rather than repeating Git's features, this description focuses on the differences between Git and Mercurial. Mercurial was developed parallel to Git by Matt Mackall - in fact its development started only a few days later, with the same intention to serve as a VCS for the linux kernel development. Git was chosen for this project, but Mercurial is used by many other projects[8]. The majority of Mercurial's code is written in Python. Python is currently developed using Subversion, but prepares to migrate to Mercurial for version control. Due to its platform independent code background, Mercurial runs on almost any platform.

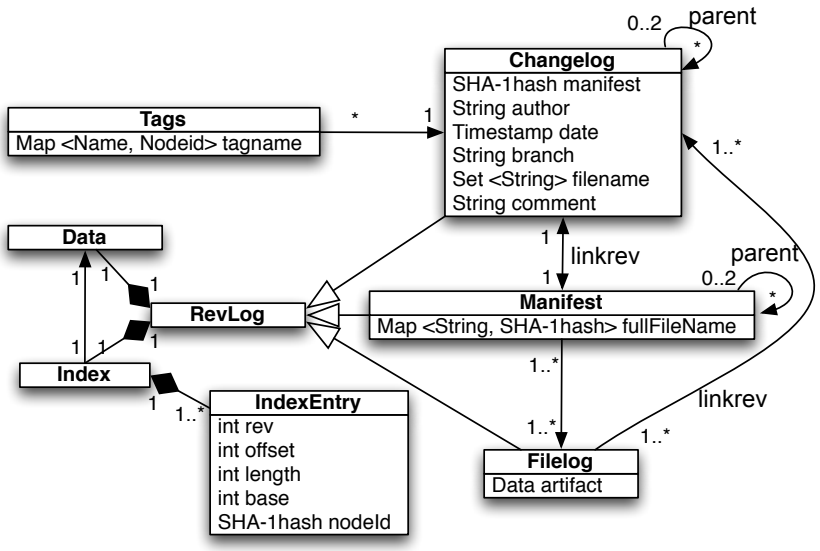


Figure 18: Metamodel of Mercurial's version model

Although Mercurial's functions are identical to Git's functions, the version model is different. Figure 18 presents Mercurial's version model. The most important metadata files are all *revlogs* (detailed further in [Maco6]). There are additional files for some information, such as the ".hgtags"-file, which stores the tags of a repository.

A *revlog* is Mercurial's equivalent to Git's *object*. It consists of two functional units, *index* and *data* file. Only when these units exceed a certain size they are separated into two single files. The *data* file stores the content of all ***versions*** of a *revlog*. The content is stored using ***delta compression***: the initial version is stored completely, while only the changes that form the next version are added. A specific version is retrieved by taking the latest full version stored, which

6 http://www.cygwin.com
7 like with Subversion, enabled by the extension hgsvn (http://pypi.python.org/pypi/hgsvn/)
8 like Mozilla (http://www.mozilla.org), OpenJDK (http://openjdk.java.net), OpenOffice (http://www.openoffice.org), SymbianOS (http://www.symbian.org), Xen (http://www.xen.org), Adium (http://adium.im), Growl (http://growl.info), Netbeans (http://www.netbeans.org), vim (http://www.vim.org), to name a few

is older than the queried version, and apply the subsequently stored ***deltas***, until the queried revision is reconstructed. Whenever a new version is smaller than the sum of all deltas, which are necessary to reconstruct this version, it is stored completely instead of as a delta. This sequence of full versions and deltas results in a very efficient space usage. Additionally, the *data* and *index* files are compressed. Therefore the space required for a Mercurial repository is only slightly larger than for a Git repository [Git], but significantly smaller than a Subversion repository.

A version in a *revlog* is identified by an identifier, called *nodeid*. Similar to the identifier in Git, it is computed by calculating a ***SHA-1 hash*** over the content (of the version, not the file, which could be the full version or a delta) concatenated with the *nodeid* of the version's parents. The parents *nodeid* is included to make the identifier unambiguous, as the same version could exist in a different place in the history. Due to using hash values of the content of an artifact to identify its version, the history is robust to manipulation, as discussed for Git. The *index* file contains the necessary information to extract a specific version from the *data* file, as the versions are stored directly after one another. There is a line (represented by the *IndexEntry*) for each version in the *index* file: the parameter *rev* is the version's local revision number, which is increased for each new entry. N.B. This order may not be the same in a ***cloned*** repository as it depends on the order the individual snapshots where stored (which can differ when storing snapshots in parallel ***development lines***). The *offset* and *length* identify where the version's data is stored in the *data* file, so that a single read access can extract it. The entry for *base* indicates the last fully stored version upon which the subsequently stored deltas are based on.

To retrieve a specific version, the *data* file needs to be read from the base version onwards. The *linkref* is an additional link to the *changelog* entry, where the *revlog* version was changed. The *nodeid* identifies the version globally, meaning that a specific version in a revlog has the same *nodeid* independent in which repository it is contained or when it was added. There are potentially two parents of a *revlog* version, whose *nodeid* is stored under $p1$ and $p2$. In Mercurial only two branches can be merged, as in most version control systems.

The *filelog* stores the versions of an artifact. A *filelog* is implemented by a *revlog* and the content stored in the *data* part of the *revlog* is the actual content (or a delta of it) of the represented artifact. A *filelog* is similar to a *blob* of Git, however all versions of a specific file are stored here, rather than in a specific version. Thus multiple *blobs* in Git can be represented by one *filelog* in Mercurial. The advantage is that the history of a single file can be searched very quickly. The name of the controlled artifact is stored in a *manifest*, which lists all filenames connected to the *nodeid* of their version. A version of a *manifest* represents a snapshot. If a user commits his changes, all files in the working copy are listed, but only the *nodeids* of the changed files are updated. Mercurial does not track versions of folders. The filename entry consists of the complete path to the file, thus changes to folders (like renames or added/removed files) are indirectly tracked by tracking the evolution of the files. A version of a *changelog* is similar to a *commit* in Git. It has entries for the *nodeid* of a manifest (i.e., snapshot), the authors name, a timestamp, additional comments, a branch name and a set of the artifacts, which were changed in order to identify which version was created the associated *manifest* version is requested. N.B. A *commit* in Git corresponds conceptually to a snapshot, while a version of a *changelog* corresponds to a change set.

As mentioned before, tags are stored in a special file (".hgtags"), which is tracked like any other artifact. Branches are managed by storing their name in an attribute of a *changelog* version. They are as lightweight as their counterparts in Git, as branches and tags are only metadata, which refer to stored data, instead of copying the artifacts versions. Artifact versions, which are created in different branches, are stored in the same *filelog*. Their order depends on the order in which the versions were applied, thus two *filelogs* representing the same artifact in two identical repositories might differ.

In Mercurial, artifact renames have to be commanded using "hg rename" explicitly. Renaming a file creates a new *filelog*. The first version entry in this new *filelog's data* file refers to the last version of the old *filelog*. The old *filelog* is no longer updated, but remains to provide older

versions of the artifact. In fact, the rename operation performs a copy, marks the copy to be tracked and removes the original artifact from being tracked.

An artifact copy can be created with or without using Mercurial's built-in command. If the copy was made without Mercurial's copy command, Mercurial will treat the copy as a complete new file. The *nodeid* of the copied version is different, because the parent *nodeids*, which are included in the calculation of a *nodeid*, are different. If Mercurial's command "hg copy" is used, Mercurial knows about the connection between the artifact's versions. Just as in the case of a renamed artifact, a new *filelog* is created, which refers to the original version's *filelog*.

Changes made to the artifact's copies follow each other, providing that the artifacts are in different repositories. A practical example motivates this behavior: Alice copies artifact *a* from the folder *test* to the folder *dev*. She did not share this change with Bob. Bob edits artifact *a*, which is still in the folder *test* in his repository. After Bob and Alice exchanged their changes Alice finds Bobs modifications in both copies of *a*. In this way the same result is preserved, as if Alice's copy and Bob's changes occurred in opposite order. If Alice renamed the artifact instead, the benefit of this behavior is more clear: Bob's changes under the old name are applied to the renamed artifact in Alice's repository.

Mercurial follows the philosophy of sharing the same repository. A locally created branch is, by default, shared with other developers. A temporary experimental branch could be created by cloning a repository (which creates a ***shallow copy*** and takes no additional space) and create a branch in this new clone. If the changes in this experimental repository should be applied to the main repository they can be pulled. If changes are shared using the pull and push operations, the repositories of the involved developers are identical. Manipulation of the history is possible with extensions, but not intended. The philosophy of Git is to develop a project in different repositories. Only chosen branches are shared, the history should be rewritten to have a clean ***development line***, etc. Using existing extensions Git and Mercurial can be used in any workflow described in Section 4.1.

5.3 PEER-TO-PEER VERSION CONTROL

Lately, a few **peer-to-peer based version control systems** emerged, which combine the serverless operation of dVCSs with the ***global repository*** of cVCSs. Peer-to-peer is well established for file sharing applications, such as BitTorrent[9] or eMule[10] and many more. In a file sharing application immutable items are offered and downloaded by other users in sophisticated ways. A file is only retrievable as long as users are online, offering this file. Thus it is not important to know which files might become available once the storing users rejoin. Typically, the files are replicated in an unstructured way, whereby peers who retrieve a file, even partially, offer it to other peers. While exploiting various load balancing, search and retrieval mechanisms, data consistency or coherency does not play a role in file sharing systems. A file is identified by calculating a ***hash value*** over its content. Thus it is not expected to be modified - changes would create a new item, which is not linked to the unmodified version in any way. Concurrent modifications can occur simultaneously and lead to contradictory items - which are also not identified as belonging together in any way. Because a shared item is practically immutable, all copies are coherent and consistent.

Based on the initial approaches for file sharing, some distributed peer-to-peer based storage systems were created. The most popular among them are Amazon's Dynamo (aka S3) [DHJ+07], Oceanstore [KBC+00] and Ivy [MMGC02], which implement a file system semantics. Contrary to the file sharing applications, stored items are expected to be modified concurrently. They are stored at specific peers - regardless of whether the peers' users intended to retrieve an item or not. Most of these systems operate on a block level basis rather than a file level. Concurrent modifications are performed by sequentializing them with the help of a single primary maintainer or a group of maintainers, which agree using a quorum vote

9 http://www.bittorrent.com/
10 http://www.emule-project.net/

[MR98] for a sequential order (like in Oceanstore). Some systems, such as Bayou [TTP+95], bring concurrent write requests in a sequential order by calculating globally unique sequence numbers (using timestamps). Depending on the system, concurrent modifications are either automatically merged (where conflicting changes are overwritten with the latest modifications) or the latest modification replaces all earlier changes. ***Outdated*** updates are prohibited nevertheless. If an artifact has already been replaced by another user's changes, all proposed changes that are based on previous versions are rejected. The latest modifications of the artifact in question needs to be retrieved and the proposed changes must be applied again.

Stored artifacts are replaced by updated artifacts. Some systems, such as Oceanstore [KBC+00], also keep the previous version, but other than this no history of the applied changes is kept (i.e., only one previous version is stored). No system handles modifications on a collection of items atomically, which would create a ***snapshot***. They guarantee ***ACID*** properties for single files only. All systems take care that coherency among the stored items is preserved. Depending on the system, the guaranteed consistency degree can range anywhere between ***eventual consistency*** and ***sequential consistency***.

There are only a few version control systems, which utilize peer-to-peer communication to develop a decentralized system. There are a number of wiki engines, which feature version control in a limited way - by overwriting articles without keeping a history of changes. The most mature approaches are presented in the following. Among these systems, only one commercial version control system exists. This system, Code Co-Op, does not rely on a peer-to-peer infrastructure, but its components communicate in a peer-to-peer fashion. To the author's best knowledge there are only four systems solely designed for version control, which are fully decentralized and built on a peer-to-peer network. These systems are described in Section 5.3.5 and the following sections. All systems are research prototypes. None of their authors seem to be aware of any other peer-to-peer based version control system, indicated by the related work discussion. The most mature system is Pastwatch, described in Section 5.3.5.

All approaches can be separated into replicated or maintainer based repository distribution. In the **replicated repository distribution** every participant has an identical copy of the entire repository. Updates on an artifact's content are distributed to all copies of the repository. Thus they guarantee ***eventual consistency*** only, but are able to control snapshots. The **maintainer based** systems divide the repository among some specific machines. Updates are sent and applied only to them. When the latest versions are to be retrieved, these distributed maintainers need to be requested. They offer ***sequential consistency***, but do control only versions of single artifacts, unable to record changes to multiple artifacts (snapshots) in an atomic fashion.

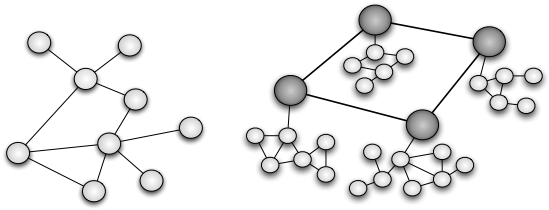
5.3.1 *Wooki*

Unstructured Peer-to-peer Overlay Network

In an **unstructured peer-to-peer overlay network** every peer keeps contact to a limited number of random peers. The network topology is illustrated in Figure 19a. The ***underlay*** network address, which is typically the ip address of these peers, is stored in a rooting table. Using the memorized underlay address, another peer can be contacted. Peers have no identifier and cannot be addressed directly. Instead requests are forwarded until they reach a peer who is able to reply. Due to the random connections and message dissemination techniques it is not guaranteed that a request will be received by the intended peer.

Wooki [WUM07] is a wiki engine which is built upon an ***unstructured peer-to-peer overlay network***. Following the replicated repository approach, each peer stores all versions of all wiki articles. When a user creates or changes an article, all changes are ***flooded*** to all other peers, line by line rather than as a complete file.

The order of the linewise update messages cannot be guaranteed, thus a later change might arrive before an earlier change was received. The so called *woot* algorithm, described in [OUMI06], ensures that changes are applied in the order in which they were made. An



(a) unstructured (b) hierarchical

Figure 19: Unstructured and hierarchical peer-to-peer overlay network topologies

update message contains contextual information such as the neighboring lines around the changed line. A received change will only be applied to a locally stored article, if all referred context lines are present. A peer stores the changes that a user performs when he is offline and propagates them when he is back online. In this time he requests all changes he missed from a neighbor to be up to date again.

The version control mechanism in Wooki does not track a version history - only concurrent updates are handled. Concurrent changes are automatically merged in the same way for each peer. However, conflicting changes, like those on the same line, are not solved. To maintain ***coherency*** conflicting changes would have to be resolved on each storing peer in the same way, resulting in identical artifacts, which cannot be guaranteed.

Propagating the update messages to all peers ensures ***eventual consistency***, but ***flooding*** in an unstructured network does not guarantee to reach all participants. Thus the multiple copies of the repository might not be coherent. The lack of coherency is not acceptable for a version control system.

Flooding

Flooding is a technique to disseminate a piece of information in an ***unstructured peer-to-peer overlay network***. Messages are broadcast in the overlay network using different techniques, which are based on receiving and forwarding a message, so that a hugh number of random peers are reached. Flooding with a time to live (TTL) is a prominent technique; A peer sends a message to all its neighbors. The receiver decrements the TTL number and forwards the message to its neighbors, if the TTL number is not zero. Upon forwarding, a peer attaches its address so that a receiver can update its rooting table.

5.3.2 *DistriWiki*

Hierarchical Peer-to-peer Overlay Network

In a **hierarchical peer-to-peer overlay network** multiple topologies are combined. In Figure 19b two unstructured network topologies are combined. The dark circles represent peers that are part of both networks. They are called **superpeers**. The white circles represent normal peers, which reside in disconnected subnetworks. Messages are routed hierarchical in both networks: First a message is ***flooded*** though the subnetwork from which it originates. The participating superpeer forwards the message to the other superpeers, which then flood them into the subnetwork they are part of.

DistriWiki [ML07] relies on the ***hierarchical peer-to-peer overlay network*** JXTA [Dui07]. **JXTA** consists of unstructured subnetworks and a structured superpeer network, which are hierarchically combined as presented in Figure 19b. In JXTA all machines in a local area network (LAN) form a subnetwork, while only the ***superpeers*** are connected over a wide area network (WAN). All superpeers in JXTA can be identified with a number. Each superpeer is responsible for storing all *advertisements*, whose identifier is numerically close to their own

identifier. An advertisement is an XML file in JXTA, which ***indexes*** a resource. DistriWiki uses it to store the name and other information (author, keywords, etc.) of a wiki article in it. Additionally to this describing information the ***peer-to-peer network identifier*** of the peer, where the respective article can be retrieved from, is stored here as well. The superpeers know at least the contact address of all superpeers, whose identifier is numerically close to their own identifier. All superpeers know each other as well. When a new article is created it is assigned a random number. Articles are stored by the peer whose user created the article. It creates a JXTA advertisement (a xml file), which contains information about the article, including which peer is offering it. This advertisement is sent to other peers. The send protocol works in a hierarchical order: The Broadcast-Mechanism of the underlying network is used to send the advertisement to all peers in the local area network (LAN). One of these peers, who has connection to other peers in a wide-area network (WAN), is the superpeer. A received advertisement is forwarded to the superpeer, whose identifier is numerically close to the article's identifier. This peer and its numerically closest neighbors store the advertisement.

When a user requests an article, which is not in the local cache, the machine looks for a local cached advertisement of that article. If nothing is found the peer sends a broadcast-message ("discover") to all peers in the LAN which is forwarded by the superpeer to other peers in the WAN. As an answer, it receives all published advertisements. As all metadata of the article are stored in an advertisement, the matching advertisement can be found. To receive the sought article, the underlying network-address of the publishing peer is extracted from the corresponding advertisement file and it is contacted directly.

As only the advertisements but not the articles themselves are replicated, only the original publisher of an article can offer it. If it goes offline, the article is unavailable as well. Like a server-based system, it represents a *single point of failure*, although only some articles (the ones published by this peer) become unavailable.

The system uses maintainer based repository control, whereby all articles are distributed among specific peers. A version history is not kept; updates overwrite the former article's content. Concurrent updates might be solved by the only peer storing the article, although this behavior is not described by the authors. Changes to multiple articles cannot be applied in an atomic transaction.

Having only a single version of an article on a single machine ensures ***coherency***, as no contradictory copy exists, and ***sequential consistency***, as the latest value is always received. The *retrievability* of articles is very limited, and overwriting changes with the latest received updates is not acceptable behavior for a version control system.

5.3.3 *CVS over DHT*

Borowsky et al. [BLS06] present an inspiring approach to a peer-to-peer based version control system. The main focus of this work is to show how traditional client-server based systems can be modified to use peer-to-peer communication. The system is based on the ***structured peer-to-peer*** system BambooDHT [RGRK04], whose development is discontinued since 2006.

The system distributes the repository in a ***maintainer based*** fashion. A complete CVS server is installed on each peer. By assigning an ID to each artifact it is distributed among the peers following the normal ***Distributed Hash Table (DHT)*** approach.

A peer is responsible for controlling the evolution of all artifacts that have an identifier numerically close to its peer ID. The peer's CVS system provides mechanisms for a file version control. This results in a truly distributed repository, in which each peer has a small set of files under control.

As in CVS, the approach is unable to support version control for snapshots. It provides ***sequential consistency*** for each artifact. ***Coherence*** is never compromized, as the maintainer of an artifact handles concurrent changes in the same way as in CVS.

Unfortunately it lacks in practical usage: user management is nearly impossible, as a user account would have to be setup on every peer. Forgery (e.g., using multiple accounts) is

Structured Peer-to-peer Overlay Network

In a **structured peer-to-peer overlay network** all peers have a unique identifier, called ***peer id***. This number is either assigned randomly or by calculating the ***hash value*** of another identifier, like the MAC address of a machine. Each peer knows the ***underlay*** address of n peers, whose identifier is numerically close to its own identifier. The metric that determines the closeness varies among different protocols. The highest value is defined to be adjacent to the lowest value, so that the identifier range forms a circle. The topology is often presented as a ring, as shown in Figure 20. The circles represent peers, which are connected through their neighbors. Contact between neighboring peers remains active by sending periodic messages. In contrast to ***unstructured peer-to-peer overlay networks***, a peer can be addressed directly. A message is sent to one of the known peers, whose identifier is closest to the intended recipient. This peer forwards the message in the same manner until the intended peer receives the message.

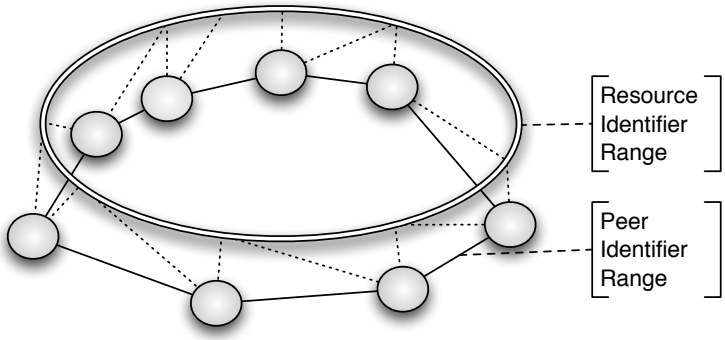


Figure 20: structured peer-to-peer overlay network topology

difficult to avoid. New artifacts cannot be detected in the system. A user needs to know a file's identification when he plans to retrieve it.

5.3.4 *Code Co-op*

Code Co-Op is a commercial distributed version control system, which was developed in 1997 and presented in [Mil97]. The version controlled repository exists as an identical ***replicated*** copy on each participating machine. The machines communicate with each other directly in a serverless fashion, by sending direct messages to each participant. Upon committing a change to form a new version, an update-script is sent to all other developers, via e-mail or local network shared folders, instead of a peer-to-peer overlay network. The self proclaimed denomination to be peer-to-peer based is not true, neither in the way the network infrastructure is maintained (e-mail and LAN), nor how the messages are distributed (from one to all other machines). The machines of the other developers acknowledge the change automatically, if no concurrent change to the committed file was received earlier. In this way an update is valid only if acknowledgement messages from all other peers are received. A second message notifies all other participants to apply the changes locally. This process is basically a two phase commit [ML83]. Occurring concurrent commits are rejected and need to be proposed again, after solving possible conflicts.

Code Co-Op provides ***eventual consistency*** only, although most of the time the provided consistency is as high as ***sequential consistency***. Commits are only applied, when no concurrent changes are committed before all participating machines are reached. The final message, which tells the other machines to apply the new version, is received at different times, thus a user might have a newer version than another user, who has yet to receive the "ok to apply"

Distributed Hash Table (DHT)

The mechanism with which items are stored among the peers of a ***structured peer-to-peer overlay network*** is called a **distributed hash table (DHT)**. An identifier is applied to an item in various ways, e.g., by calculating the ***hash value*** of its content or name. This identifier and the ***peer IDs*** are taken from the same value range. This can be ensured by calculating the identifier with the same ***hash function*** with which the peer IDs were calculated.

A peer is responsible for storing the items with the numerically closest identifier to the peer's ID. The closeness is determined by the same metric with which the distance between the peers is measured. Figure 20 shows how the identifier ranges are related to each other: The dotted lines indicate that a peer is responsible for all resources with an identifier that is equal to or greater than its peer ID, up to the next greater peer ID. N.B. A resource can be anything which can be shared, including storable items, hardware resources or offered services.

To provide a higher degree of availability of the shared resource, the direct ***neighbors*** in the key space of the responsible peer replicate the resource, and are named **replica peers**. On updates they must update their resource as well. If the responsible peer fails, the ***routing mechanism*** automatically routes requests to the next closest peer, which is one of those neighbors.

message, resulting in ***sequential consistency***. However, messages can be lost completely, or machines could be present without the knowledge of the committing machine. If in these circumstances a participant does not receive an update which is accepted by all other machines he misses the latest version. Upon receiving a later version a repair mechanism has to retrieve the missing versions. In this situation only ***eventual consistency*** is provided. Lost messages or missing or unexpected participants are, however, unlikely, as Code Co-Op is based on reliable transport mechanisms and assumes stable user machines. Moreover, failing machines or lost messages can lead to situations, where the replicated repository is in an ***incoherent state***, e.g., when two concurrent changes are applied in two different temporary ***network partitions***.

The protocol effectively implements a two phase commit protocol [ML83], which can blockade if messages are lost. Code Co-Op seems unsuitable for a wide distributed or large user group, as the more messages there are to be sent, or the longer the messages need to reach the participants, the more likely concurrent updates are. And if concurrent updates occur, all updates have to be repeated, thus extending the commit operation.

5.3.5 *Pastwatch*

Pastwatch [YCM06] is a peer-to-peer based version control system, which is the result of a PhD thesis at the Massachusetts Institute of Technology [Che04] in 2004.

Pastwatch's basic structure is shown in Figure 21. Each user works on artifacts in a ***working copy***. Changes form a ***snapshot***, which is stored locally and on other peers in a ***structured peer-to-peer overlay network***, using the ***DHT*** mechanism. The used peer-to-peer overlay protocol was developed exclusively for Pastwatch, although any structured overlay, which offers the services of a DHT could have been used. Pastwatch's implementation, however, does allow that only a group of peers are offering the DHT service. The offering peers can be a smaller subgroup or even dedicated machines, which are not used by a user for Pastwatch. Weaker machines or machines with a highly dynamic network connection are supported in this way, as they can participate without offering the version control service to other peers. In this way a concrete deployment can be optimized for the network traffic: The greater the number of DHT serving peers is, the more messages, due to routing and retrieving stored items, are produced. But the smaller the number of DHT serving peers is, the more dominant the drawbacks of centralized systems (see Section 2.4.1) are. Beside the GNU diff [GNUa] tool, every functionality was built from scratch.

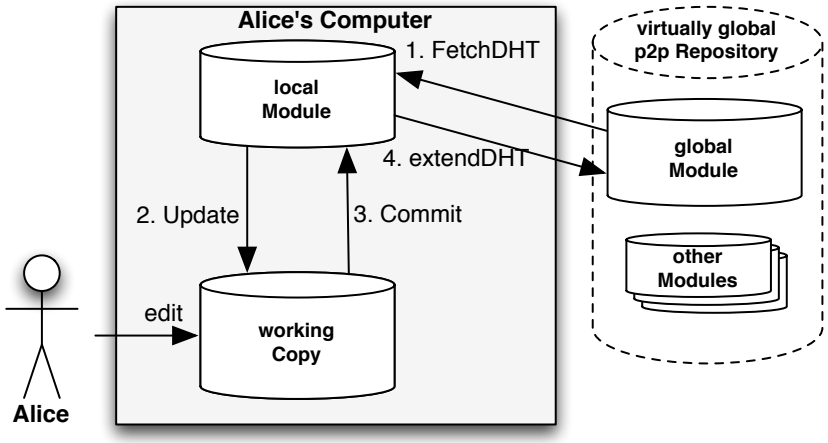


Figure 21: Basic structure of Pastwatch's repository

A snapshot contains only the changes made to all artifacts since the last snapshot, along with the following metadata: The author's identity, consisting of his name, a ***public key*** and the identity of the previous snapshots the current snapshot was based on. There could be more than one previous snapshot if multiple snapshots were ***merged***. Following the DHT mechanism, a snapshot is stored by the peer, whose identifier is numerically close to the snapshot's hash value, which is calculated using a ***hash function*** on the snapshot's content. Changes form a new snapshot, so the already stored snapshot is never overwritten. For each global module a membership list exists in the DHT, which is created by an administrator. It lists all users, along with their name and ***public key***. The list is ***signed*** using the administrators public key and stored on the peer, whose identifier is numerically close to the value computed using a hash function on the administrator's public key.

Public Key Cryptography

Public key cryptography is a mechanism for encrypting and signing data. A user creates an asynchronous key pair, a **public key** and a **private key**. They are generated using a large random number. The private key must be kept secret while the public key can be distributed among other users.

Data can be **encrypted** using a public key. Only with the respective private key the data can be decrypted. This can be used, e.g., for delivering secret messages. The message is encrypted using the receivers public key.

A message or an artifact can be **signed** by encrypting it with one's own private key. Any user can decrypt it with the respective public key. Doing so proves that the person having the corresponding private key encrypted the message or artifact.

The information stored last is a list called **log head**, which stores a list of the identifiers of all snapshots committed by a specific user, signed with the user's public key. This list exists for every user and is stored in the DHT using the calculated hash value of the user's public key. By doing so, only the corresponding user can update that list. In this way this ***index*** of the snapshots of every user is written only by the respective user, which avoids concurrent updates, which could overwrite each other. Combining all log heads would form a complete index of the repositories snapshots.

When a user updates his working copy he requests the latest snapshots using Pastwatch's *fetchDHT* operation: First the membership list is retrieved to find out all current members of the module. Next, all ***log heads*** of these members are retrieved from the DHT. Comparing all listed snapshots in the retrieved *log heads* to the snapshots in the local module identifies the missing snapshots. They are retrieved from the DHT in the final step of the updating process. Now the working copy can be updated to the latest snapshot in the local module.

Due to disturbances in the network, resulting in lost messages or disconnected peers, possibly forming a disconnected partition, some snapshots might be not retrievable. To keep the local module ***coherent***, snapshots are only applied if the snapshots that they are based on are at hand. Additionally, the signature has to fit to the author's public key. The local module is updated periodically and before each commit operation. If the latest *membership list*, all *log heads* and all missing snapshots could be retrieved, a local module is identical to the virtual global module.

Local changes are published by committing all changes in a user's working copy with a snapshot to the local repository. The operation *extendDHT* ***signs*** the snapshot, stores it in the DHT and creates a new entry in the *log head* corresponding to the committing user. By storing snapshots, which are unique immutable items for each author and version, and having the per author *log head* only written by a single author (multiple reader, single writer), concurrent write operations cannot overwrite data and do not need to be synchronized, which is a challenging task in a peer-to-peer overlay network.

Most of the time the version history of all snapshots is linear. If not all snapshots can be fetched or snapshots are concurrently submitted, an ***unnamed branch*** is created. They are detected only by a user who issues the fetchDHT command after the diverged snapshots can be retrieved. The authors of Pastwatch recommend to merge unnamed branches as soon as they are detected. The following example should clarify the situation in which a concurrent commit results in an unnamed branch: Alice and Bob changed files in their snapshots. These changes could be non conflicting, e.g., be on a disjunct set of artifacts. Alice retrieves Bob's *log head*, which only contains the old snapshots' identifiers, which are in Alice's local module. Before she stores her snapshot in the DHT, but before she can add its identifier to her *log head*, Bob commits his snapshot. He fetches Alice's snapshot, which contains only old snapshot referrals as well. Not knowing about Alice's latest snapshot he submits his snapshot. The local module of Alice and Bob is not identical to the virtual global module, as the other's latest snapshot is missing. Any user executing the update operation will receive both latest snapshots, which coexist, both being based on the same snapshot. This user's development could continue by basing his changes to Alice's or Bob's or both snapshots. In the latter case the unnamed branch is merged.

If network distortions lead to the situation whereby the peer and its replicating peers, which store a specific version are no longer reachable, a peer which executes the *fetchDHT* operation will notice the missing snapshot. If the snapshot is in its local module it reinserts it in the DHT. It is not likely that all storing peers fail simultaneously before they can update replacing peers. But if the network falls into disjunct partitions all those peers might be in one partition, unreachable to the peers in the other partition. When one of those peers reinserts the missing snapshot, and the partitions join each other, the stored snapshot is identical, and can be overwritten by any copy. A ***log head*** is only modified by one user. If it is stored in a disjunct partition a new *log head* is created. Each user keeps a local copy of the ***log head***, thus previous entries are not lost.

Note that the authors of Pastwatch did not consider the situation where concurrent commits result in unnamed branches. They claimed that this can only happen, when the network is falling into partitions. Assuming that in each partition only one unnamed branch could be created, they argued that the diverging snapshots would be discovered when the disjunct partitions are reunited and then, being a small number, can be easily merged. But as illustrated in the previous example, concurrent commits can result in unnamed branches in the same partition. They are detected by the next user, who updates his changes, who may not be one of the previous authors of the affected snapshots. The periodical updates might lead to a situation whereby nearly all members of a module discover diverging snapshots at the same time. If any two authors merge these snapshots, even by applying the same changes, they might form two new diverging snapshots, which differ only in the author. The more members a module has, or the further away these members are from each other, the more likely it is that unnamed branches will be created. Users might lose interest in reconciling them, as the merged snapshot might be concurrently committed and result in a new unnamed branch.

There are some **hotspots** in Pastwatch, i.e., peers whose services are used more frequently than by other peers. As these peers can only serve a specific number of peers at any one time, the scalability of Pastwatch is seriously limited. These peers are all peers who store a ***log head*** and the one peer that stores a membership list. Whenever any peer executes the update operation all this peers are contacted. The membership list is first queried for all members, whom's log heads are retrieved next, to check for a new snapshot. Besides being initiated by a user the update operation is also performed periodically (every two minutes according to the authors' recommendation). The commit operation triggers the update operation as well.

The evaluation presented in [YCM06] does not confirm our scalability concerns and showed instead that the time needed for the update operation to complete grows very slowly for a rising number of members, it takes on average only 1.9 seconds more time to update a module consisting of 200 members compared with updating the same module having only 2 members. However, in this setup the number of DHT serving peers was constant to eight peers, and only two peers actively contributed new versions. The number of users was raised to 200, but it is unrealistically to assume that those users do not execute any operation and all share the deployed ten machines.

To summarize, the hotspot peers, which need to be contacted all the time, the high number of parallel messages involved in this operations and the high possibility to create unwanted unnamed branches are the biggest drawbacks of Pastwatch, which limit its scalability drastically.

Pastwatch manages concurrent commits ***optimistically***, by allowing parallel versions forming an unnamed branch, which should be merged as soon as it is detected (which is not enforced). In order to have all latest versions all storing peers forming the ***DHT*** needs to be contacted. Each peer returns the identifier needed to retrieve the versions created by a single user. Only ***eventual consistency*** is guaranteed, as messages might be received later or may be lost completely. During a ***network partition*** or due to ***indeterministic routing***, which can occur under high ***churn***, some versions might be not retrievable at all, when the storing machines are in another partition. If all peers, who store a user's *head log* (the maintainer and its replicating peers) are in an unreachable partition, the location of the latest snapshots cannot be retrieved, even if they are stored in the current partition (only, if a snapshot based on the missing snapshot exists and can be retrieved, the identifier of the otherwise missing snapshot can be retrieved from its metadata). If the *log head* is accessible in the other partition, the identifier of the missing snapshot can be looked up, but the snapshot cannot be retrieved (as it is stored in the other partition).

It seems to be an impractical decision to separate the storage of a user's log head and her created snapshots. In the described cases, where some versions cannot be retrieved, the virtual ***global repository*** is ***incoherent***, although Pastwatch recovers from this state, once the partitions rejoin. A more serious situation happens, when the membership list gets updated on more then one maintainer, due to the mentioned network disturbances (network partitioning or routing inconsistencies.). The authors of pastwatch did not described, how it can recover from this situation. We assume that no conflicting updates can occur (a user reaches only one maintainer, to whom's list he joins or leaves), thus the diverging list could be automatically merged.

5.3.6 *GRAM*

In [TM04a] and [TM05] a so called Group Revision Assistance Management (GRAM) is presented, which is a prototypical implementation based on the ***hierarchical peer-to-peer overlay network JXTA***. The version control mechanism only reuses the GNU diff tool [GNUa], everything else is built from scratch. The system does offer only single file revision control, omitting snapshots and branches or tags. Revisions in the repository are stored as ***forward deltas*** only. Another limitation is that only textual files can be handled by the system.

GRAM follows the ***replicated repository distribution***, where each peer has a locally stored repository, which is identical to all repositories stored by other peers. When a version is created, a message is broadcast to update the repositories stored by the other peers.

Conflicts are ***resolved proactively*** using an interesting new approach. Rather than locking edited artifacts, a peer sends a message with the changes made to the currently edited artifact to all other peers. These other peers check, whether the received modifications conflict with the modifications made by their user. If a conflict is detected both users are informed and can contact each other via an integrated instant messenger. In this way conflicts are detected before a version is committed. The mechanism is called **hybrid concurrency control**, because it resolves conflicts proactively like the ***pessimistic concurrency control*** but allow parallel work on the same artifacts, similar to the ***optimistic concurrency control***. The detection mechanism works on a granularity of lines in an artifact. To enable this mechanism GRAM has to be used with the provided editing environment.

However, high message delays or overloaded peers can delay or drop messages, which can lead to a situation whereby existing conflicts are not detected in time. If the network falls into partitions, conflicting versions cannot be detected. GRAM does not offer a mechanism to reconcile conflicting versions, which can be created under the mentioned circumstances.

GRAM offers ***eventual consistency***. The obvious drawback is the high and frequent message transfer. Whenever any user edits an artifact, a broadcast message is routed to all other peers. This limits GRAM's scalability. In a network with high message delay, as being likely in a ***GSD*** scenario, GRAM might not work properly. It can occur that later versions arrive at a peer before versions they are based on are received. This leads to an ***incoherent*** repository at the affected peers.

5.3.7 *SVCS*

In [LLST05], a simple version control system, called Scalable Version Control Layer (SVCL), is presented. Peers communicate using the ***structured peer-to-peer overlay network*** Chord [SMK+01b, SMLN+03], which offers the ***DHT storage mechanism***.

Similary to CVS over DHT (presented in Section 5.3.3) the repository is distributed in a ***maintainer based*** manner: According to the assignment calculated following the ***DHT*** approach, an artifact is maintained by a specific peer. These peers control the evolution of the respective files and stores all versions. There is no other repository on a user's machine. Concurrent updates are ***controlled pessimistically***, by locking artifacts. The offered operations are limited to the basic commandos offered by GRAM (see Section 5.3.6): Committing changes to single files. Non-textual files can be handled as well.

A workflow in SVCS is as follows: A user locks an artifact he aims to change. When he finished his modifications he commits them to the maintaining peer, who stores them in a new version and releases the lock.

As in the approach presented in Section 5.3.3, artifacts can only be detected, if their identifier is known. The ***pessimistic concurrency control*** limits concurrent work, but guarantees a ***coherent*** repository at all times. SVCS provides ***sequential consistency*** with respect to single files only. However, in a peer-to-peer network, machines can fail at any time. If a peer who acquired a lock on an artifact fails, the artifact cannot be modified by anyone else. Using a timeout mechanism to release a lock, if the acquiring peer does not respond is also critical, as the peer might still be online. If messages are lost or a peer indeed failed cannot be destinguished by other machines in the network. The authors of SVCS did not address this problem.

5.3.8 *Chord based VCS*

In [JXY06] a system is presented, which uses ***structured peer-to-peer overlay network*** Chord [SMLN+03] as well. The offered version control functionality is similar to that of CVS (see Section 5.1.3), supporting branches and tags, but being limited to single file revision control.

The repository is stored both, ***replicated*** among all peers, and distributed using the ***DHT*** storage mechanism. While the artifacts are stored distributed in the DHT, the metadata, including the artifacts location and all version control information, is stored replicated on all peers on the following way: All actual artifacts are chopped up into units, which can be transferred in a single network message, which might be an ethernet frame, although this was not further clarified by the authors. Each unit is identified by a random number. Using this identifier, all units are stored distributed among the peers by the DHT mechanism. The metadata of an artifact refers to all units of an artifact. The metadata itself seems to be stored in identical copies on all peers. Changes are recorded as **operation-based diffs**. Rather than identifying changes made to the content of an artifact, as in the **state-based diff** approach, the operations which lead to the changes are recorded. The metadata contains this information along with other information that describe a version (like the author's name, the version number, etc.). It appears that the metadata is stored as identical copies by all peers. A new version is shared by sending its metadata to all participants, dividing the new version into small units and storing these units in the DHT (by sending them to the responsible peers).

Similar to GRAM (see Section 5.3.6), conflicts are meant to be ***resolved proactively*** using a ***hybrid concurrency control***. When a user edits an artifact, all other users who edit the same artifact, are informed, probably by a broadcast message, but it might be coordinated by a single maintaining peer as well. This is not detailed by the authors, however, when changes are committed they are sent to all peers instead of a maintaining peer, which lead to the conclusion that there is no maintaining peer in the system. A version can only be committed, if all possible conflicts are resolved.

The Chord based VCS seems to be very preliminary, although a prototypical implementation is described but not evaluated. Future improvements were promised, but not published at the time of writing.

The proactive conflict detection mechanism raises the same problems as discussed in Section 5.3.6 and does not promise to be a valid solution in a peer-to-peer based system. Since updates are broadcast to all peers, later versions could be received before the versions they are based on, which guarantees ***eventual consistency*** only. Lost messages or ***network partitions*** prevent peers to contact each other and conflicting versions are not noticed, leading to an ***incoherent*** repository.

5.4 SYSTEMS FOR COLLABORATIVE WORK SUPPORT

There are numerous client-server based tools that support ***global software development (GSD)***. Most of them cover only a particular aspect of GSD, sometimes integrated into a single platform, e.g., IBM Rational [IBM]. Jazz [HCRP04] is a tool environment, intended specifically for GSD. It is based on the client-server paradigm. Jazz evolved from the Eclipse IDE [Fou]. It features a notifying system, which broadcasts general messages in the project, for example when a task is completed. There are a number of other tools integrated, such as a change management system. The version control, however, is managed by ClearCase (presented in Section 5.1.5).

According to the authors' best knowledge, there is no other integrated tool environment that fulfills the needs listed in Section 3.2.

Groove Virtual Office [Gro] is a collaboration environment that is partly peer-to-peer-based. In its first version it had serious scalability problems, barely supporting 20 developers in the same workspace [Har01]. When Microsoft bought Groove Networks in March 2005 in order to save the project [Mil], it became evident that the technology was still not ready for the market. The software was restructured to improve scalability using the client-server approach. The current version fulfills several security requirements but still does not support version control management.

5.5 SUMMARY

The version control systems presented in this chapter can be categorized into ***centralized***, ***distributed*** and ***peer-to-peer based*** version control systems.

We chose to present the most important centralized version control systems from the numerous existing solutions. The presented systems are the most widely used or present a major milestone in the evolution of version control systems.

The distributed version control systems rose to prominence in 2003 with the implementation of *Monotone*. There are significant fewer distributed systems available than centralized solutions. Nevertheless, especially *Git* and *Mercurial* are as mature as their centralized counterparts, which have been in use for many years. The two latter systems substituted centralized version control systems for a large number of (open source) projects.

Directly related to our solution are the peer-to-peer based systems, which we presented exhaustively. There are, to our best knowledge, no further peer-to-peer based version control systems, as a productive system or as a research prototype.

We preliminary discussed that none of the presented systems fully satisfy the requirements of globally distributed software development. The centralized solutions bring unbearable delays for some developers, the distributed systems are unable to share the produced snapshots effortlessly with all developers. The peer-to-peer based solutions tend to solve these issues but cannot provide a reliable degree of consistency. We analyze the presented approaches and their implemented design decisions in detail in the next chapter.

6 ANALYSIS OF RELATED VERSION CONTROL SYSTEMS

In this chapter we analyze the properties of the examined systems described in the previous chapter. We designed our version control system, PlatinVC, based on our analyses of the architecture and its impact on the properties of the related approaches.

The first section describes, which of the requirements introduced in Chapter 3 are met by the individual systems. We analyzed which system properties influenced satisfying those requirements in Section 6.2. In Section 6.3 the systems are examined regarding the minimum provided consistency degree. This chapter is concluded with a taxonomy of the analyzed peer-to-peer based approaches (Section 6.4), and a summary of promising building blocks (Section 6.5).

In the following comparison tables the prototype developed in this thesis, PlatinVC, is analyzed as well. How requirements are fulfilled or how certain properties are achieved is detailed in the next part of this work.

6.1 REALIZED REQUIREMENTS PER SYSTEM

Table 1 lists all requirements, which were introduced in Section 3.2. "✓" indicates that a requirement is fulfilled, "✗" marks that a requirement is not met, "–" indicates that the requirement is not applicable to the system, and "⦸" denotes requirements, which are partially fulfilled. The background details, which lead to this analysis, can be found in the systems descriptions in Chapter 5.

Requirement \ System	RCS	SCCS	CVS	Subversion	ClearCase	ClearCase Multisite	Monotone	Git/Mercurial	Wooki	DistriWiki	CVS over DHT	Code Co-Op	GRAM	SVCS	Chord based VCS	Pastwatch	PlatinVC
Functional requirements																	
R-1: Support common version control operations																	
R-1.1: Support parallel work on all artifacts	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓
R-1.2: Detect concurrent modifications	–	–	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	–	✓	✓	✓
R-1.3: Resolve concurrent modifications	–	–	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	–	✓	✓	✓
R-1.4: Track single artifacts in an ordered history	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
R-1.5: Track snapshots in an ordered history	✗	✗	✗	✓	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗	✗	✓	✓
R-1.6: Retrieve a recorded state	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓	✓	✗	✗	✗	✓	✓
R-1.7: Choose recorded versions of selected artifacts	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✗	✗	✓	✗	✗	✓
R-1.8: Individual working copies	✗	✗	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓	✗	✓	✗	✓	✓
R-1.9: Enable local commits	✗	✗	✗	✗	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓
R-1.10: Track variants (branches)	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗	✓	✓	✓

Continued on next page

Table 1 – continued from previous page

Requirement \ System	RCS	SCCS	CVS	Subversion	ClearCase	ClearCase Multisite	Monotone	Git/Mercurial	Wooki	DistriWiki	CVS over DHT	Code Co-Op	GRAM	SVCS	Chord based VCS	Pastwatch	PlatinVC
R-1.11: Tag with labels	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗	✓	✓	✓
R-2: Change the name or path of artifacts	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	✗	✓	✓	✗	✓	✓	✓
R-3: Support for multiple projects	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✓	✗	✓
R-4: Conjoined local repositories	✗	✗	✗	✗	✗	✓	✗	✗	✓	✗	✗	✓	✓	✗	✓	✓	✓
R-5: Redundant backups	⊘	⊘	⊘	⊘	⊘	✓	⊘	⊘	✓	✗	✗	✓	✓	✗	✓	✓	✓
R-6: Interoperability of different VCSs	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓
R-7: Connect from anywhere	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	✗	✗	✗	✗	✗	✗	✗	✗	✓
R-8: Offline version control	✗	✗	✗	✗	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓
R-9: Local commits	✗	✗	✗	✗	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓
R-10: Traceability	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
R-11: Easy to setup/manage	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
R-12: Prevent censorship	✗	✗	✗	✗	✗	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
Non-functional requirements																	
R-13: ACID																	
R-13.1: Atomicity	✗	✗	✗	✓	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗	✗	✓	✓
R-13.2: Consistency	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗	✗	✗	✓	✓
R-13.3: Isolation	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
R-13.4: Durability	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗	✗	✓	✓
R-14: Availability	✓	✓	✓	✓	⊘	⊘	✓	✓	✓	✓	✓	✗	✓	✗	✓	⊘	✓
R-15: Transparency	–	–	–	–	–	✓	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
R-16: Minimal storage space consumption	✓	✓	✓	✓	✓	✓	✓	✓	✗	–	✓	✓	✗	✗	✗	✗	✓
R-17: No centralized services	✗	✗	✗	✗	✗	✗	–	–	✓	✓	✓	✓	✓	✓	✓	⊘	✓
R-18: Adequate communication speed	✗	✗	✗	✗	✗	⊘	⊘	⊘	✗	✓	✓	✗	✗	✓	✗	✓	✓
R-19: Scalable and robust																	
R-19.1: Robust to highly fluctuating users	✓	✓	✓	✓	✓	✓	✓	✓	⊘	✗	⊘	⊘	⊘	⊘	⊘	⊘	⊘
R-19.2: Robust to shifting users	✓	✓	✓	✓	✓	✓	✓	✓	⊘	✗	⊘	⊘	⊘	⊘	⊘	⊘	⊘
R-19.3: Scalable to a growing number of users	✗	✗	✗	✗	✗	✗	✓	✓	⊘	✓	✓	⊘	⊘	⊘	⊘	⊘	✓
R-20: Continuous expendability	✗	✗	✗	✗	✗	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
R-21: Fault tolerance	✗	✗	✗	✗	✗	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
Security Aspects																	
R-22: Access and usage control	⊘	⊘	⊘	⊘	⊘	⊘	–	–	✗	✗	✗	✗	✗	✗	✗	✗	⊘
R-23: Keep sensitive artifacts	✓	✓	✓	✓	✓	⊘	–	–	✗	✗	✗	✗	✗	✗	✗	✗	✗
R-24: Attribute based access control	⊘	⊘	⊘	⊘	⊘	⊘	–	–	✗	✗	✗	✗	✗	✗	✗	✗	⊘
R-25: Confidentiality	⊘	⊘	⊘	⊘	⊘	⊘	–	–	–	–	–	–	–	–	–	–	⊘
R-26: Data integrity	⊘	⊘	⊘	⊘	⊘	⊘	✓	✓	⊘	✗	✗	⊘	⊘	✗	⊘	✓	✓

Continued on next page

Table 1 – continued from previous page

Requirement \ System	RCS	SCCS	CVS	Subversion	ClearCase	ClearCase Multisite	Monotone	Git/Mercurial	Wooki	DistriWiki	CVS over DHT	Code Co-Op	GRAM	SVCS	Chord based VCS	Pastwatch	PlatinVC
R-27: Non-repudiation	⊘	⊘	⊘	⊘	⊘	⊘	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓
R-28: Authentication	⊘	⊘	⊘	⊘	⊘	⊘	–	–	✗	✗	✗	✗	✗	✗	✗	✓	⊘
R-29: Secure communication	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	✗	✗	✗	✗	✗	✗	✗	✗	⊘
R-30: Low administration overhead	✗	✗	✗	✗	✗	✗	⊘	⊘	✗	✗	✗	✗	✗	✗	✗	✗	⊘

Table 1: Fulfilled requirements (see Section 3.2)

6.2 ANALYSIS OF SYSTEM PROPERTIES LEADING TO FULFILLED REQUIREMENTS

In this section we look into the system properties that enable a system to fulfill or not to fulfill a requirement.

6.2.1 *Functional Requirements*

R-1: Support common version control operations *(R-1.1 and R-1.11)*

Every examined system, except DistriWiki, fulfills some of the sub-requirements *R-1.1* and *R-1.11* of requirement *R-1*. Whether these requirements are fulfilled does not depend on a system's architecture but solely on the implemented features.

R-2: Change the name or path of artifacts

Only two approaches (*CVS over DHT* and *SVCS*) fail to fulfill requirement *R-2*. In both approaches the repository is stored partitioned among maintaining peers. Other systems, which fulfill this requirement, also store a partitioned repository, but in a different way: The two problematic systems identify the repositories parts, which are the individual artifacts, by the artifact's name, i.e., the maintainer of the artifact calculates its ***ID*** by ***hashing*** the artifact's name. The maintainer stores an artifact's complete history and manages the access to it. However, if the artifact is renamed, the calculated hash value changes, hence a new peer becomes responsible for that artifact. If only the old name of the artifact is known, under which the versions where stored, the latest versions are not retrievable (i.e. their new location is unknown). If only the new name is known, all versions which where committed before the artifact was renamed are not accessible. This problem could be solved by storing a reference to the new location on the old maintainer and vice visa. However, both approaches miss to do so.

R-3: Support for multiple projects

This Requirement can be fulfilled in all of the analyzed architectures.

R-4: Conjoined local repositories *and* R-5: Redundant backups

All systems which fulfill these requirements store a local repository on each machine. Requirement *R-4* cannot be fulfilled, if a the repository is stored partially, because the parts locally missing would have to be retrieved from distant repositories. Similarly, requirement *R-5* presumes that all versions are in one place, which can be backed-up. Complete repositories, such as the ones on client-server systems, can be backed-up using other tools.

R-6: Interoperability of different VCSs

Fulfilling requirement *R-6* depends on implementation only, as, theoretically, any two version control systems can be combined. However, when two version control systems interoperate, only the version control features offered by both systems can be used. It is, for example, not possible to record snapshots, when CVS and Git are used interoperablely.

R-7: Connect from anywhere

In order to *connect from anywhere* network obstacles, such as ***NAT***, have to be circumvented. This could be achieved by using supporting tools or setups, such as virtual private networks (VPNs).

R-8: Offline version control

Requirement *R-8* is fulfilled by providing a local repository on the machine, where new versions can be submitted, before sharing them with all other participants. A distributed system, which can overcome ***network partitioning***, offers an even better offline support. Here a user can publish her new versions, as if they were shared with other participants. She forms a network partition of her own. A repair mechanism is automatically sharing her versions, once she is online again.

R-10: Traceability

Support for traceability does not require any specific system properties and could be implemented in all of the analysed architectures. An implementation is, however, challenging as pointed out by Section 7.3.7.

R-11: Easy to setup/manage

Client-server based solutions require expertise and time to get running. As they are crucial for the system they must be maintained well. Peers in a peer-to-peer based system are almost self-maintaining. Periodical maintenance mechanisms detect and repair disturbances in the network. This is necessary because the system does not rely on any of its machines. Thus it is not important to diligently maintain an individual peer. Additionally, a peer is never used solely for the purpose of the system; normally a participant uses his machine for other tasks as well. Thus maintenance of the individual machines is carried out by its respective users.

R-12: Prevent censorship

The analysis of the examined systems clearly shows that only machines which have a single copy of a repository cannot *prevent censorship*. In all distributed systems the content is stored widely spread and redundant across all or a number of randomly chosen machines. Hence all copies in the system in all locations (as well on currently offline machines) would have to be altered to censor the content of an artifact.

6.2.2 *Non-functional Requirements*

R-13: ACID*:* R-13.1: Atomicity

Atomicity is only applicable to the systems, which fulfill requirement *R-1.5: Track snapshots in an ordered history*. For these systems, providing requirement *R-13.1* is the basis to fulfill requirement *R-13.2: Consistency*. Atomicity can be achieved by various techniques. Changes made by a user can conflict with an already applied ***snapshot***. In a centralized system this can be checked in one place, but if the repository is distributed multiple machines have to decide, whether a change can be applied.

Code Co-Op (see Section 5.3.4) uses a transaction protocol, known from database systems. In a two phase commit protocol all participants agree or disagree to accept a proposed snapshot. A transaction requires many messages. Additional messages might be needed when the participating machines are less reliable. Additionally, the transfer times of messages in a peer-to-peer network can vary widely, which makes it difficult to set realistic time-outs.

Only one peer-to-peer based version control system supports snapshot version control and guarantees atomicity: Pastwatch (see Section 5.3.5). Instead of a sophisticated transaction protocol, which decides whether a snapshot is accepted, all snapshots are accepted. Identical to the mechanism used in the distributed version control systems snapshots that are ***based*** on the same base version form an ***unnamed branch***, as detailed in Section 5.2.1.

R-13: ACID*:* R-13.2: Consistency

Some systems do not guarantee ***coherency*** in all situations, thus they cannot guarantee ***consistency***. If we correlate the degree of consistency stated in Table 2 with the basic architecture of the different systems listed in Table 3 we can identify which mechanism influences a certain ***consistency degree***. The more centralized the management of the repository is, the higher the achieved consistency degree is. There is no centralized version control system, which has a worse consistency than ***sequential consistency***. In these systems a user either receives the complete requested history or nothing at all. The distributed version control systems provide ***causal consistency***. Here either the complete history of a participant or nothing is shared. The complete history of a participant is, however, not the complete history among all participants (the global history), but a small part of that history. This part includes all versions made by a single user, and all of the versions that these versions are based on.

When examining the peer-to-peer based version control systems we see this trend continuing. Only systems, in which specific peers maintain the complete history of individual artifacts, following the ***maintainer based repository distribution*** discussed further in Section 6.5.2, offer ***causal consistency***. A single message can contain all missing parts of the history of a single file. Retrieving a projects history, which is tracked using snapshots is, however, challenging - none of the examined maintainer based peer-to-peer version control systems support this. The only supporting maintainer based system, Pastwatch, does not store the history of individual artifacts on a maintainer, but a per-user list of snapshots instead (for more details see Section 5.3.5). A single message can only contain an incomplete history, in which versions created by other users are missing. Hence Pastwatch provides ***eventual consistency*** only.

R-13: ACID*:* R-13.3: Isolation *&* R-13.4: Durability

In traditional peer-to-peer based content distribution a peer does delete locally stored content prior *joining* the network. This avoids the presence of outdated content. The versions in a version control systems are, however, never ***outdated***. Thus a peer does not have to delete previously stored versions, as they never become obsolete. He has to take care to update to the latest version only. Pastwatch and Code Co-Op are the only systems doing so. Nevertheless, the main effect is to reduce the time needed to update a joining peer, as the redundant version history should be also stored in the network.

R-14: Availability

Nearly all industry proven version control systems are able to run under the major operating systems, being Mac OS X and similarly Unix systems, Windows, and Linux. Notable exceptions are Code Co-Op, which runs in Windows only, ClearCase, which does not support Mac OS X, and Git, which is difficult to set up in a Windows environment.

In spite of the fact being research prototypes, many peer-to-peer based systems are platform independent. Those systems are all implemented using the Java programming language. SCVS

was implemented using C and runs on Linux only, and Pastwatch was created using C++, compiled only for Linux, Mac OS X and other Unix systems.

R-15: Transparency

Being applicable to distributed systems only, all peer-to-peer based systems satisfy this requirement. Managing the connection transparent is an inherent property of peer-to-peer systems, fulfilled by the ***DHT mechanism*** as well. The distributed version control systems are the only ones, in which the users are required to specify where to obtain updates from.

R-16: Minimal storage space consumption

Rather than storing a full artifact for each version delta compression can be used, which stores the differences between the versions only. Over the years multiple sophisticated variants have been developed, such as wave deltas, forward deltas, etc., which optimize the trade-off between storage space and access time of an artifact's version. A noteworthy exception is demonstrated by Git, which stores full artifacts in all versions, and compresses the entire repository afterwards. Benchmark tests in [Git] showed that the storage space consumption of Git is the smallest compared to Subversion and Mercurial.

Minimize the repository storage using deltas is implemented by nearly all non peer-to-peer based version control systems, but only two of the peer-to-peer based systems. These two systems are maintainer and replica based, proving that requirement *R-16: Minimal storage space consumption* is not dependent on the basic architecture.

R-17: No centralized services

Almost all peer-to-peer based systems are faithful to their promise not to rely on a central service. Only Pastwatch has a central service: The member list of a module is stored an a maintaining peer and its replicating peers. Every time a peer queries for new updates (and additionally periodically) this membership list has to be retrieved to check for new members. Being stored by the ***DHT mechanism*** this list is replicated and does not depend on the presence of a specific peer, but as long as the currently maintaining peer is online it has to serve all requests.

R-18: Adequate communication speed

As detailed in Section 2.4.1 client-server based systems are not suitable for globally distributed development. The server remains at a fixed location, which introduces long transfer delays for all users, who are physically distant to the server. ClearCase Multisite (see Section 5.1.5) offers a solution to this problem, by distributing the repository among servers, which are located close to the developer groups. These servers are configured to permit write access to a disjunct repository part only. Scheduled maintenance synchronizes the distributed repository, but in between these periods users have to cope with long delays, if they need to read the latest versions or write to parts of the repository, which are on a distant server. The distributed version control systems do not aim to have a global repository status stored on any participant's machine. Here users exchange their repositories directly, using one of the transport media listed in Table 3. It depends on the location and used medium how big the delay is.

Examining the different peer-to-peer based systems shows that ***replication based*** systems cannot provide adequate access times. While old versions are locally present new updates are flooded in the network, taking time to reach every participant. To avoid an ***incoherent*** repository all distributed versions have to be acknowledged, before they can be applied, which requires the exchange of multiple messages in a commit protocol. In a ***maintainer based*** system any peer might store the latest versions. The delay to this peer might be very small, if this peer is physically close, or large, if this peer is located distant. On average the communication delay is the median of all delays among all participating peers. The delay can

be minimized using content distribution techniques like ***owner path caching*** [CSWH00] or proactive content placement [ATS04].

In a globally distributed development scenario, as presented in Section 2.3, joining and leaving users have a positive side effect called ***shifting***. In the running example in Section 2.5 company *CA* is located in Berlin, Germany and company *CB* is located in Kolkata, India. Between these locations there is a 4.5 hour time difference. When the developers of *CB* in India end their daily work in 5 p.m. the employees of *CA* in Germany are just coming back from lunch, as it is there 1.30 p.m. local time. Whenever a peer goes offline its stored repository is replicated to a substituting peer. That way the repositories stored on peers in India are shifted to peers in Germany, as the number of peers in India are shrinking and the number of peers in Germany are rising, wherefore it is more likely that a vanishing peer in India is replaced by a joining peer in Germany. However, if the transition is too rapid, the replication mechanism could be not fast enough, resulting in lost repositories. Requirement *R-19.2: Robust to shifting users* formulates this case.

R-19: Scalable and robust

Only the distributed version control systems are not affected by robustness or scalability issues when users are joining or leaving in large numbers, as they can be used between two users at a time only. The client-server architecture is not affected by a changing number of users (requirement *R-19.1*) or regardless of whether a group of users disappear in one location and another group joins in second location (requirement *R-19.2*). As detailed in Section 2.4.1 servers can only support a rising number of users, if their hardware is extended.

Peer-to-peer overlay networks are inherently strong in providing a good scalability, supporting requirement *R-19.3: Scalable to a growing number of users*. Participants, who use an offered service bring new resources to offer a service themselves. It depends upon the concrete system implementation, which services can be offered by how many peers. All except two (DistriWiki and CVS over DHT) of the examined systems have a centrally offered service, which is much more frequently accessed than other services. The scalability in these systems is limited, as in one point the few peers, who are offering the popular service, cannot serve the demand.

The more peers disappear, the more outdated the contact information on a peer are. More and more messages are sent to peers, which are no longer existent. Whenever a peer joins, it needs to obtain up-to-date routing contacts, whenever a peer fails, it needs to be replaced. The more frequent or the higher in number these changes are, the worse the peer-to-peer overlay network can handle them, resulting in delayed or lost messages. If a peer providing a particular service fails, this service would become unavailable. Thus the service is ***replicated among a fixed number of peers***. In the ***replicated repository*** based systems all peers are able to offer the systems services, making them very robust. The ***structured peer-to-peer overlay networks*** have a lesser degree of robustness.

Services are offered by individual peers are addressable using the ***DHT*** mechanism. A ***replica peer*** takes over the responsibility of a failed peer. When a huge number of peers fail in one region and new peers appear in another region, the new peers may act as replica peers and take over the duties of the failed peers. This shift of responsibility in the ***DHT*** mechanism automatically transfers all needed data and offered services to the appearing peers.

R-20: Continuous expendability

Client-server based systems become unavailable during an update of the software or hardware of the server. The distributed and the peer-to-peer based systems can update parts of the system, while other parts take over offered services, using the replication mechanism described in the last paragraph. If updates to the system are distributed partially they need to be compatible with the remaining, previous versions.

R-21: Fault tolerance

Again, using the replication mechanism, failing peers can be countervailed. In systems, which do not replicate offered services, faults are fatal.

6.2.3 *Security Aspects*

While there are proven security solutions for centralized systems most aspects are still the subject of current research for distributed systems. In a client-server system it can be assumed that attackers are among the clients only (ignoring the ability to compromise the server, when it can be physically accessed). In a distributed system anyone can be an attacker - especially in a peer-to-peer system, where everybody stores data and offers services as well.

In all of the examined systems versions belonging to a specific artifact are stored among random peers. It is necessary to be able to choose any peer as a ***replica peer***, in order to not worsen the robustness, scalability and availability. Thus requirement *R-23: Keep sensitive artifacts* seems unable to be realized by peer-to-peer bas ed systems.

The requirements *R-22: Access and usage control*, *R-24: Attribute based access control* and *R-25: Confidentiality*, which all deal with access control, are challenging to implement in ***peer-to-peer networks***. The access and account information has to be stored integrally and confidentially.

Requirement *R-26: Data integrity* is partially fulfilled in the ***replicated repository distributing*** systems, as each peer has a copy of the secured artifact. If any peer were to alter the content of some artifact it could be noticed by comparing it with the other versions. A different mechanism, introduced by the dVCSs, calculates the ***hash value*** of some artifacts. Any manipulation would lead to a completely different hash value, if the users identifier is included in the metadata, and the hash value of this metadata is stored requirement *R-27: Non-repudiation* is met as well. The systems fulfilling these two requirements (*R-26* and *R-27*) by calculating the hash value of each version, which is included in a snapshot, where the authors identity is also stored. The hash value calculated using this snapshot is stored in the predecessor, so that a cryptographically save history is created.

There is no fully decentralized mechanism that completely satisfies requirement *R-28: Authentication*. Usually a central certificate authority is needed initially to map a user's real identity to a certified virtual identity and verify his virtual identity whenever demanded by other users.

All of the examined systems are able to fulfill requirement *R-30: Low administration overhead* or requirement *R-29: Secure communication* - which are implementation issues only. However, when fulfilling requirement *R-29*, **recursive routing**, where a message is forwarded to the next closest receiver, cannot be used, as the destination field in the message have to be encrypted as well. Using **iterative routing**, where a peer informs the requesting peer about a closer peer, instead of forwarding a message by itself, would solve this problem. Alternatively, only the message itself could be encrypted.

6.3 DEGREE OF CONSISTENCY

The presented version control systems differ in the degree of consistency that they provide. Table 2 lists the systems, sorted by the minimum guaranteed consistency degree, as presented in Section 4.4.2. The stated guarantees are valid in worst case scenarios, where a system might be handicapped, and only a partial update is delivered, as analyzed before. If no update is received at all, it is not considered as a violation of the guaranteed consistency degree.

We can see that almost all systems using ***maintainer based repository distribution*** offer the high consistency degree of ***sequential consistency***. Only Pastwatch, which is the only peer-to-peer based version control system capable of handling snapshots, does provide ***eventual consistency*** only.

System	Consistency Degree	*Sequential Consistency*	*Causal Consistency*	*Eventual Consistency*	*Always coherent*
SCCS (Section 5.1.1), RCS (Section 5.1.2)		⊘			✓
CVS (Section 5.1.3)		⊘			✓
Subversion (Section 5.1.4)		✔			✓
ClearCase (Dynamic View) (Section 5.1.5)		✔			✓
ClearCase (Snapshot View) (Section 5.1.5)		✔			✓
ClearCase Multisite (Dynamic View) (Section 5.1.5)			✔		✓
ClearCase Multisite (Snapshot View) (Section 5.1.5)			✔		✓
Monotone (Section 5.2.2)			✔		✓
Git (Section 5.2.3)			✔		✓
Mercurial (Section 5.2.4)			✔		✓
Wooki (Section 5.3.1)				⊘	✗
DistriWiki (Section 5.3.2)		⊘			✓
CVS over DHT (Section 5.3.3)		⊘			✓
Code Co-Op (Section 5.3.4)				✔	✗
GRAM (Section 5.3.6)				⊘	✗
SVCS (Section 5.3.7)		⊘			✗
Chord based VCS (Section 5.3.8)				⊘	✗
Pastwatch (Section 5.3.5)				✔	✓
PlatinVC (Section 7.3.6)		⊘	✓		✓

(Legend: regarding ⊘= single artifacts, ✓ = snapshots, ✔ = single artifacts & snapshots)

Table 2: Guaranteed degree of consistency

6.4 TAXONOMY OF KEY MECHANISMS

Table 3 concludes the analysis of the related systems by identifying how certain key aspects are solved. The ***cVCSs*** and ***dVCSs*** are summarized in one entry each. All peer-to-peer based approaches are listed.

6.4.1 *Concurrency Control*

Concurrent updates can be controlled by three classes of mechanism: pessimistic, optimistic and hybrid **concurrency control**.

The more restrictive ***pessimistic concurrency control*** was developed first. Conflicting versions are proactively avoided by granting only one editor write access, while multiple other users are only allowed to read the artifact in question. Traditionally a ***locking mechanism*** has been used, where a developer has to acquire a lock for an artifact he plans to edit.

The more advanced classical ***optimistic concurrency control*** allows artifacts to be edited in parallel, as it assumes that most modifications do not conflict. Multiple authors copy an artifact, modify its content, and merge the resulting versions, resolving conflicts that might have occurred. We refer to it in Table 3 with *optimistic (merge)*.

Two mechanisms were developed, which are hybrids between optimistic and pessimistic concurrency control. One approach (named **hybrid resolving**) tries to detect when two developers begin to modify the same artifact. The authors are informed so that they can coordinate their changes immediately. The second mechanism (**hybrid voting**) allows artifacts to be modified concurrently, but hinders committing them, if concurrent versions are to be submitted. Before each commit a message is sent to all other participants. Only if they agree, indicating that they did not changed the artifact in question, the artifact can be committed as well.

6.4.2 *Repository Distribution*

The repository is either stored ***replicated*** or ***maintainer based***. A replicated repository exists as an identical copy on each participant's machine. The VCS needs to take care to update all repositories once artifacts have been changed. If the repository is stored maintainer based it is separated into parts and distributed among specific machines, which store the parts and handles any updates on them. The partially stored repository parts are usually not identical, and in order to obtain the latest version of all artifacts in the repository all maintainer have to be queried. How the parts are distributed, i.e., how many parts are maintained by which machines, is subject to the distribution mechanism and detailed in the description of the respective system in Chapter 5. All of the presented peer-to-peer based approaches store the partial repository using the ***DHT mechanism***. This means that a part can be addressed by an identifier, which is computed using information about the part.

In traditional peer-to-peer based file sharing the identifier is obtained by calculating the ***hash value*** of the stored content with a cryptographic save hash function, which results in a unique identifier (see [RFH+01]). Only if the content of two copies is identical they do have the same identifier. Theoretically calculating the hash value using different content could result in the same value, but the possibility for this to happen is unrealistically small. The ***asymetric key cryptography*** is based on the premise that the resulting hash values are in practical usage unique [Eck09]. To calculate an item's identifier its content has to be known - which would mean it has been received already. Therefore an additional ***indexing*** mechanism is needed, which lists the stored items identifier (on a central server, e.g., in the eDonkey network [HBMS04] or BitTorrent [Coh03]). Storing an artifact by storing its identifier on one peer, and its content on a different one, is called **indirect replication**.

A stored item could be addressed by an identifier, which is calculated using a meta information of the item, like its name. This could be the filename of a file, the branch

name of a stored branch, or any other assigned name. By knowing the name of the desired item the hash value forming its identifier can be computed.

6.4.3 *Repository Partitioning*

If the repository is not stored as identical replicas it is stored dissect into parts among all machines. How the parts are distributed is subject to the distribution algorithm. Among the examined VCSs the repository is partitioned in the following ways:

by branches: A machine stores all versions of all artifacts, which belong to the same branch. In the traditional workflow practiced by the users of a cVCS branches are created to keep track of the main ***development line*** to implement features and to fix bugs. A positive effect is that often only updates from a specific branch are needed. If the latest versions are desired only the machine that maintains the branch of interest needs to be contacted. However, to get to know existing and newly created branches an additional indexing service is needed.

The main development line is often split into further branches, which keep track of different releases, like a future release and a current. Beside this organizational separation into branches different variants can be maintained. The branches are used with a different intensity. The machines that store the more popular branches would have a higher load. If only a main branch is frequently accessed the serving machine acts practically as a central server.

The DVCs introduced a novel approach to use branches. In addition to the outlined traditional usage, branches are used to separate the development of the individual developers. Every developer commits his created versions exclusively to the repository stored on his machine. If the default branch is used and shared with another developer, it appears conceptually[1] as a different branch in the other developer's local repository. Popular workflows using this approach are detailed in Section 4.1. The access tp the branches is naturally balanced among the participants, where most write accesses occur in the local branch. Some branches, however, might be more popular and accessed more often by other participants. Often, as presented in Section 4.1, a *blessed branch* exists, which integrates the contributions stored in other branches. The machine storing this blessed branch has to serve read requests much more frequently than other machines. In practice read access is offered by storing the blessed repository on a dedicated public server.

by user changes: In Pastwatch the repository is separated by users' changes. Similar to the way dVCSs use branches to separate the contributions of single users, Pastwatch distributes the repository into parts, which store all changes of a single user. More precisely, a list with all snapshots created by a single user (called head log) is stored by a machine. This list includes all identifiers of the machines, which actually store those snapshots. While in a dVCS the ultimate separation boarders are branches, meaning that other users' contributions are also stored there, when branches are merged. Pastwatch clearly separates the versions created by different users. Thus the stored snapshots do not reconstruct the gapless history of a branch. If a snapshot committed by a specific user was based on a snapshot of another developer, and the basis snapshot of the latter snapshot is to be retrieved, the machine storing the other user's partial repository needs to be requested. The actual snapshot is retrieved indirectly by accessing the other user's log head first, from where the location of the snapshot can be retrieved. The direct basis snapshot can be retrieved from the metadata stored within the succeeding snapshot.

Separating the repository by users brings the benefit that each user has to communicate with only one other machine, to store his changes. Only a single user sends write requests

1 This branch is even named after the developer in Git [HT]. In Mercurial [Mac] it appears as an unnamed, parallel branch.

to that machine, while all other users only read the stored information. As mentioned before this access only updates a user's log head, while the actual immutable snapshots are stored distributed among peers in the DHT. The log head is accessed by calculating the ***hash value*** of the respective user's name. An identifier of a committed snapshot, which is stored in this list, is formed by calculating the hash value of that snapshots content. A disadvantage is that a list of all users has to be available, in order to know the locations of all versions of which the repository consists.

by artifacts/folders: The complete history of an artifact could form a part. When the VCS also tracks snapshots, this information has to be stored, as it cannot be clearly assigned to a single part. All examined solutions avoid this problem by controlling the versions of single artifacts only.

Rather than storing the complete history of a file, a more corse grained approach would be to store the history of all files in a folder, without subfolders. If subfolders would be stored the part that includes the outermost folder would store the entire repository.

The advantage of this approach is that all versions of an artifact are retrievable from one machine. Using this approach, the distributed part is identified by the hash value calculated over the artifacts or folders name. By knowing the name the complete history can be accessed. This approach presents us with two challenges: When artifacts are renamed, or in the case the repository is separated by the stored folders, a folder is renamed or the artifact is moved, the identifier changes. This would lead to a new part. The old and new parts would have to refer to each other, in order to be able to traverse the history. A similarly issue are newly created artifacts or folders. When their name is unknown they cannot be retrieved. An additional announcement service would be necessary.

by snapshots: Some systems distribute the repository by storing the snapshots among different peers. If no additional ***index*** is stored, a snapshot's identifier needs to be known in order to retrieve a snapshot. A snapshot always refers to the snapshot it was based on. This information could be used to traverse to older snapshots from a recent one, but more recent snapshots could not be found. Storing a snapshot's identifier in the parent snapshot's metadata would require to alter it. Concurrent access would have to be controlled. To retrieve versions by traversing known and received versions would take a long time, as multiple sequential requests would need to be issued, instead of only one, who retrieves the latest snapshot.

It is a better approach to have an index, which lists the snapshots and their relations. This index does not have to be on a central machine but could be distributed following the approaches presented here. Pastwatch stores the history index distributed, which contains all snapshots made by one user. In GRAM the index is replicated among all machines.

Separating the actual snapshots from the place where the history index is stored delays the retrieval of a snapshot, as first a snapshots location and subsequently the snapshot itself must be retrieved.

6.4.4 *Communication Paradigm*

The communication paradigm influences a system's properties. Three different communication paradigms are used: Client-server, peer-to-peer or communication over different existing infrastructures, such as e-mail, USB flash memory drives or ad-hoc communication channels. The differences of the client-server and the peer-to-peer communication paradigms have been detailed in the previous chapters. The last approach relies on an existing infrastructure.

6.4.5 *Communication Protocol*

The various used communication protocols listed in Table 3 have a little influence on a system's properties. Their efficiency regarding the time needed to route a message is varying, but the basic properties are similar for all protocols, which belong to the same communication paradigm.

6.5 PROMISING MECHANISMS

In this section the most promising mechanisms implemented in the presented related work are analysed. PlatinVC's design was based on this analysis.

6.5.1 *Concurrency Control*

The analysis of related systems pointed out that ***pessimistic concurrency control*** introduces numerous problems into a peer-to-peer based system. A lock has to be acquired and released. As long as an artifact is locked, only one developer can modify it. This blocks parallel work on the same artifacts. The practical usage of the first VCSs (SCCS (see Section 5.1.1) and RCS (see Section 5.1.3) showed that a user tends to lock more files, than he modifies, just in case he might need to change them. The more users a system has, the higher is the possibility that an artifact is locked, which another user intended to modify. Releasing a lock is error prone in a highly dynamic environment like a peer-to-peer network. A peer who governs the lock over an artifact cannot decide, if a peer is not online anymore, otherwise all messages sent to him would be lost. In the latter case reassigning the lock to another developer would lead to the undesired situation where both developers modify the same artifact.

In conclusion, ***optimistic concurrency control*** is better suited to a peer-to-peer based VCS. It proved to be the winning model, as only a few modern VCSs implement a pessimistic concurrency control.

The two hybrid approaches sound promising - but lack in practical usage. The need to contact all other participants in order to perform a commit limits the scalability to the point where no user might be able to practically perform a commit operation successfully.

If the number of participants in the first approach (*hybrid acknowledge*) rises, the possibility that the same ***basis version*** has been modified in the time span needed to reach all participants (which also rises with the number of participants) increases. If two or more artifacts were changed based on the same version none can be committed. Both authors have to try again to submit their versions, until one of them is successful. The other one could now apply his changes to the committed version. It is, however, very likely in a big project that third developers try to submit their contributions concurrently.

The second approach (***hybrid resolving***) is more promising. Contacting all participants in a network takes again more time when the number of participants rises. But a detected conflict does not block the commit approach completely. It is resolved interactively by connecting all authors, so they can coordinate the integration of their changes. This mechanism, however, relies on the fact that all participants can be contacted, when they edit files. Requirement *R-8: Offline version control* cannot be provided. If the network falls into ***partitions*** or the routing is temporarily ***indeterministic***, the concurrently editing authors cannot contact each other, resulting in committed versions in the separate networks. When the partitions rejoin a conflict exists and has to be solved with other means. This approach is interesting as an additional feature, but unable to avoid conflicts in a peer-to-peer network by itself. Another practical problem is that the total number of participants in any peer-to-peer network is fluctuating and nearly impossible to measure exactly. Thus an author never knows, if he received answers from all participants.

The two hybrid approaches have not been pursued further, possibly because the mentioned problems remained unsolved.

System Mechanism	cVCSs	dVCSs	Wooki	DistriWiki	CVS over DHT
concurrency control	old: pessimistic new: optimistic	optimistic	optimistic	update overwrites	optimistic
repository distribution	central server(s)	arbitrary	replicated	maintainer (no replicas!)	maintainer
repository partitioning	single copy	by branches	identical replicas	by artifacts	by artifacts
communication paradigm	client-server	one-to-one	unstructured p2p	hierarchical p2p	structured p2p
communication protocol	mostly WebDAV	proprietary, http, smtp	proprietary	JXTA	BambooDHT

System Mechanism	Code Co-Op	GRAM	SVCS	Chord based VCS	Pastwatch	PlatinVC
concurrency control	hybrid voting	hybrid resolving	pessimistic	hybrid resolving	optimistic	optimistic
repository distribution	replicated	replicated	maintainer	replicated + indirect	maintainer + indirect	maintainer
repository partitioning	identical replicas	identical replicas	by artifacts	identical replicas	by users	by folders
communication paradigm	broadcast	hierachical p2p	structured p2p	structured p2p	structured p2p	structured p2p
communication protocol	ethernet, smtp, smb	JXTA	Chord	Chord	proprietary	Pastry

Table 3: Architectural aspects of the examined VCSs

The only peer-to-peer based VCS, which implements hybrid resolving combines it with an unstructured peer-to-peer network. In this class of peer-to-peer networks messages cannot be guaranteed to reach all participants. There is a chance that two messages sent from different authors reached a disjunct group of receivers only. In this case both would be unaware of the other and commit their conflicting changes.

6.5.2 *Repository Distribution*

Keeping identical replicas ***consistent*** in a peer-to-peer network is a challenging task. The problems discussed to reach all participants might hinder some machined to get an update. Due to the routing mechanism messages travel at different speeds, so that an update might be received, which is based on an still missing update. All systems, which distribute the repository as identical copies among all machines can guarantee ***eventual consistency*** only. As there is no single controlling point it is difficult to detect and solve conflicting changes in the same way in all repositories, endangering its ***coherency***.

An advantage is that updates are available without the need to query for them. Committing changes, however, takes time as all participants have to be reached.

The ***maintainer based approach*** has the advantage that concurrent commits can be resolved by a central instance, the maintainer. If two concurrent versions are committed, a maintainer receives both requests and can handle them in a sequential order. E.g., he could accept the first received commit and reject the second. However, in some situations the two commits could be received by different peers, who believe they are the only maintainer and each accept the received version, not noticing the conflict. ***Network partitions***, lost messages, and ***indeterministic routing*** can lead to such a problem. An additional problem is a performance issue. While commits can be applied very quickly, as only a single peer has to be contacted, updates are slow to be received, as multiple maintainer have to be addressed.

Replicating identical copies of the repository among all peers makes the system more robust and scalable (requirement R-19). The data integrity, a protection objective demanded by requirement R-26, is also supported, as there are numerous copies in the system. Nevertheless, both requirements can also be fulfilled using the maintainer based approach, thus the limited advantage given for these aspects is outweighed by the higher consistency degree, which can be attained using a maintainer based repository distribution. Having user maintained local repositories next to the system maintained maintainer based repositories combines the advantages of both approaches and helps in fulfilling the requirements R-4: *Conjoined local repositories*, R-5: *Redundant backups*, and R-8: *Offline version control*.

The first versions of PlatinVC followed the maintainer based approach. Later prototypes also adapt the properties of the replicated distribution, which is detailed further in the following part of this work. The basic structure of PlatinVC, however, is maintainer based.

6.5.3 *Repository Partitioning*

The repository distribution has the most influence on the ***consistency degree*** a system provides. The examined systems use multiple approaches to partition the repository. The load distribution differs among the approaches used. However, no approach has a clearly better load distribution than another. Whenever a system has a central component, which needs to be accessed frequently, the load distribution is worsened.

Created versions are immutable and can be addressed by calculating a ***hash value*** using their content. The stored versions have to be ***indexed***, i.e., the hash values have to be listed somewhere, as they cannot be calculated without the content, by an additional mechanism. The required indirection of first accessing this index and afterwards the versions themselves does not seem to bring an advantage. In contrast, if one of the two machines required (and their replicating machines) are missing, the versions cannot be retrieved. The chances that this situation can occur is doubled.

Most systems partition the repository according to the stored artifacts. The artifacts are addressed by the hash value of their name. By keeping all versions of a single artifact on one machine all versions are retrievable, whenever the maintaining machine can be contacted. This ensures ***sequential consistency***, but only regarding single artifacts and not snapshots. Having an initial version of a project files, however, enables one to update all known files to the latest versions without the need to request an additional index. However, the problem of announcing new artifacts as well as renaming an artifact (see requirement *R-2*) have to be solved in this approach.

All other approaches do not seem to be more beneficial. They bring the problem to announce new items as well. E.g., in Pastwatch the repository is partitioned according to its users. All users are listed in a centrally stored membership list. Whenever the snapshots created by a user are not accessible, gaps exist in the projects history, thus only ***eventual consistency*** can be provided. Partitioning a repository by its branches brings the same problem that older versions might not be retrievable.

Summarized, the most promising partitioning scheme seems to be the mostly used: separation according to stored files.

6.5.4 *Communication Paradigm*

The communication structure has a great impact on the features and drawbacks of the version control system. The client-server based communication suffers from the disadvantages discussed in Section 2.4.1. The centralized version control systems perform worse with a rising number of users and are prone to complete system failure. Solving some problems of the centralized approaches (like requirement *R-12: Prevent censorship*), the distributed version control systems are introducing new issues: manual effort is required in order to share changes, which therefore distribute slowly among the developers, leading to unaware duplicate development and conflicts being found late.

The peer-to-peer communication structure demonstrated to overcome these shortcomings and fulfill especially the requirements *R-11: Easy to setup/manage*, *R-15: Transparency*, and *R-17: No centralized services*. The system data and operations are distributed among the participants, where replication helps to avoid even partial system failure. Additionally, every user is free to leave the system anytime, having the independence of a dVCS with the connectivity of the cVCS.

Approaches in which messages cannot be addressed to a specific recipient, as in ***unstructured peer-to-peer overlay networks***, can only be combined with replicated repository distribution, where updates are broadcast. To overcome the limitation of ***eventual consistency*** a paradigm where a specific participant can be addressed is needed, as realized by a ***structured overlay network***. Following the analysis in Section 2.4 and this chapter, the structured peer-to-peer communication paradigm is the most promising.

6.5.5 *Communication Protocol*

We saw in the following sections that a ***structured peer-to-peer protocol*** is the most promising communication protocol. Among the available implementations Chord ([SMK+01b, SMLN+03]) and Pastry ([RD01b]) are the most widely used ones. BambooDHT (presented in [RGRK04]) is a variant of Pastry. [RD01b] showed that Pastry has the better system properties. However, there are several other structured peer-to-peer overlays.

6.6 SUMMARY

The related systems, which were presented in Chapter 5 were analyzed in this chapter. A distributed version control system, which promises a high ***consistency degree*** and fast operation times, should be a combination of the following design decisions for the identified

key mechanisms: Only the ***peer-to-peer*** communication paradigm complies with the majority of the identified requirements. All other examined communication paradigms have integral properties, which hinder the realization of certain requirements.

The combination of ***distributing the repository maintainer based*** while providing user maintained local repositories showed to be superior to ***replicated repository distribution***. Although some problems remain unsolved, partitioning a repository by artifacts, i.e., giving the control of all versions of an artifact to a maintaining peer, whose identifier is numerically close to the ***hash value*** of the artifact's name, is the most promising solution. A challenge is to handle new, renamed or moved artifacts. A greater issue is to handle ***snapshot*** based version control without introducing complex transaction protocols.

Part III

PEER-TO-PEER VERSION CONTROL SYSTEM - PLATINVC

The previous part points out that state-of-the-art version control systems are not suitable for global software development.
A novel version control system, PlatinVC, which primarily aims to support global distributed collaborative development, is presented in this Part. Chapter 7 presents this solution from the perspective of a user. Design decisions and alternatives are detailed in Chapter 8. A prototypical implementation, based on these design decisions, is presented in Chapter 9. Results from a testbed evaluation we performed with our prototype are discussed in Chapter 10.

7 OVERVIEW OF PLATINVC

The main goal of this thesis is to provide a suitable solution for version control to aid globally cooperating contributors: PlatinVC is a fully decentralized version control system, which is built on a minimal set of assumptions listed in Section 3.1 and fulfills the requirements stated in Section 3.2.

To provide a general understanding of PlatinVC, this chapter describes its basic architecture (in Section 7.1) and a typical and recommended workflow (in Section 7.2), which is distinct from the workflows practiced in other VCSs (introduced in Section 4.1). The distinguishing features of PlatinVC are presented in Section 7.3 and version control services offered to users are detailed in Section 7.4.

This chapter focuses on a user point of view and introduces the philosophy that PlatinVC was designed upon. Their technical implementation along with a discussion of alternatives are detailed in Chapter 8.

7.1 BASIC ARCHITECTURE

PlatinVC inherits some properties of the ***dVCS*** as well as the ***cVCS*** by combining aspects of their architectures. From a functional point of view, PlatinVC acts like a mixture of cVCSs and dVCSs, inheriting their positive features, while omitting their flaws, as discussed in Section 2.4.1. The basic architecture depicted in Figure 22 combines aspects of a local users components in a dVCS, as presented in Figure 14, and of a servers components in a cVCS, as presented by Figure 12.

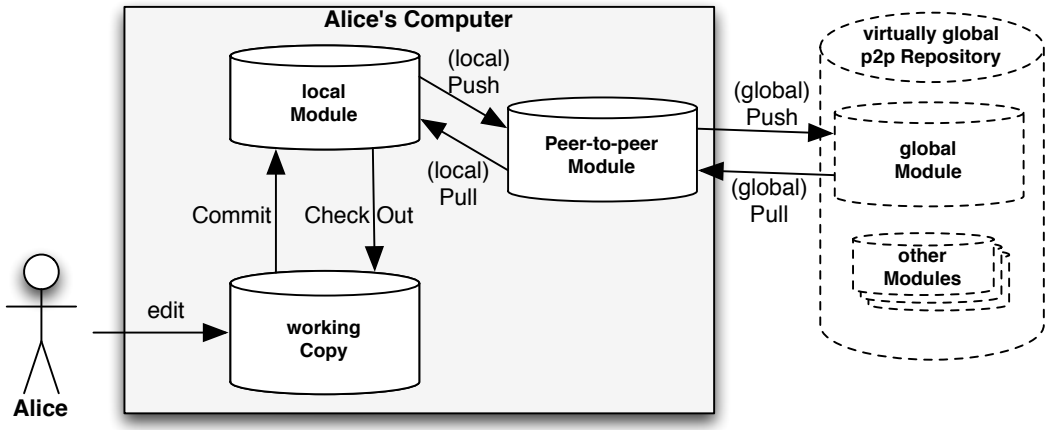


Figure 22: Basic architecture of PlatinVC

Note that the terminology introduced by CVS (see Section 5.1.3) is used, rather than the terminology common in Mercurial (see Section 5.2.4): Multiple modules reside in a repository. A module stores the committed history of all files in a working copy, which usually belong to a project.

A user works on artifacts in a ***working copy***. Typically all files in a ***working copy*** belong to the same project. A **software product** consists of one or more of these projects, depending on its organization. All changes are tracked by executing the ***commit*** operation. Whenever this occurs all modifications are recorded in a ***snapshot***, which is stored in the ***history*** of a local ***module***. When the developer decides to share his changes he executes the ***push*** command. All

changes that were recorded by the ***local module***, but not pushed before, are now pushed to the ***Peer-to-Peer Module*** first and subsequently to the respective *global module* in the virtual global repository. The peer-to-peer module acts like a (outdated) mirror of the global module in the repository. The **virtual global repository** comprises all existing modules. We call it virtual, because it does not exist on a single machine. It is, rather, distributed among the participants in a structured way, in opposition to the chaotic distribution of the modules in a dVCS. This allows for retrieval of the latest snapshots as if a centralized repository was accessed, as in a cVCS. Chapter 8 describes in detail how this is achieved.

7.2 WORKFLOW

Although any workflow presented in Section 4.1 could be used with PlatinVC, we present a recommended workflow in this section. Using this workflow exploits all benefits of offered by PlatinVC.

7.2.1 *Frequency of Commits*

As discussed in Section 4.2, there is a trade-off in using version control systems. On the one hand, snapshots should be committed as frequently as possible, to enable fine granular rollbacks. On the other hand, shared changes should not corrupt the project (e.g., resulting in an uncompilable source code project), hindering further work of collaborators.

PlatinVC inherits the solution to this problem from the dVCSs. A user can commit snapshots locally to build up a fine granular history (similar to a dVCS). When the latest snapshot reflects a stable project state, the push operation makes all committed snapshots available to all other participants, as in a cVCS.

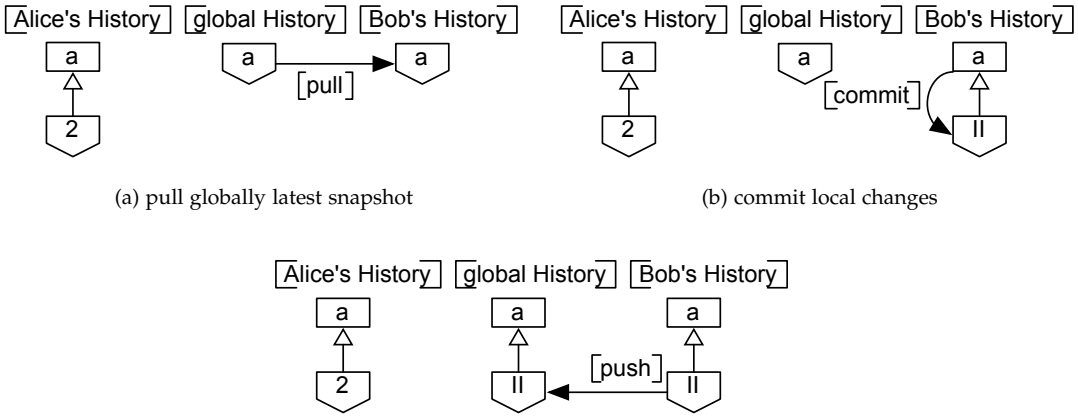
The two modules on each peer enable the feature to separate personal commits from globally shared commits. While the local module always contains all locally recorded snapshots, the peer-to-peer module includes all globally shared snapshots of all users, which where retrieved in the last update.

7.2.2 *Repository Sharing*

Any workflow, such as presented in Section 4.1, can be used with PlatinVC. All workflows for dVCS can be applied by interacting with the local module only, which is a true dVCS repository as well. A centralized workflow can be applied by interacting with the virtual global repository only. Therefore after each commit execution, a push has to be invoked. All changes would then pass through the local module and become immediately visible to coworkers.

PlatinVC was designed to suit the following workflow, presented in Figure 23. All figures show an abstract view of PlatinVC on two machines and the virtual global repository. The history of Alice's local module is presented on the left, the history of Bob's local module is presented on the right. The globally visible history in the corresponding global module is presented in the middle. All other components failed to maintain conciseness. Named branches are not shown either. For simplicity we assume that there exits only one named branch in this example. A square with a number represents a recorded snapshot, the square with a triangle attached being the latest snapshot. The arrows with the white head connecting snapshots represents the reference from a snapshot to its parent. The arrows with the black heads represent executed operations.

1. Before developing a new functionality, a developer updates the project he plans to work on, by pulling (see operation *O-6: Pull globally*) the changes from the virtual global module into the local module (see Bob in Figure 23a). Therefore, the developer's machine contacts a few other machines from whose peer-to-peer module the updates are retrieved. The update could cover all branches or specified branches of the complete working copy or specified folders with their contents (artifacts and subfolders) only. He now updates



(a) pull globally latest snapshot

(b) commit local changes

(c) push snapshot globally

Figure 23: Basic Workflow in PlatinVC

his working copy with the latest snapshot of the branch he intends to work on. As explained in Section 5.2 there could be multiple heads in a branch - a developer chooses one to base his changes on.

The project should be in a stable state, so that the developer can be sure that only his modifications can destabilize the project.

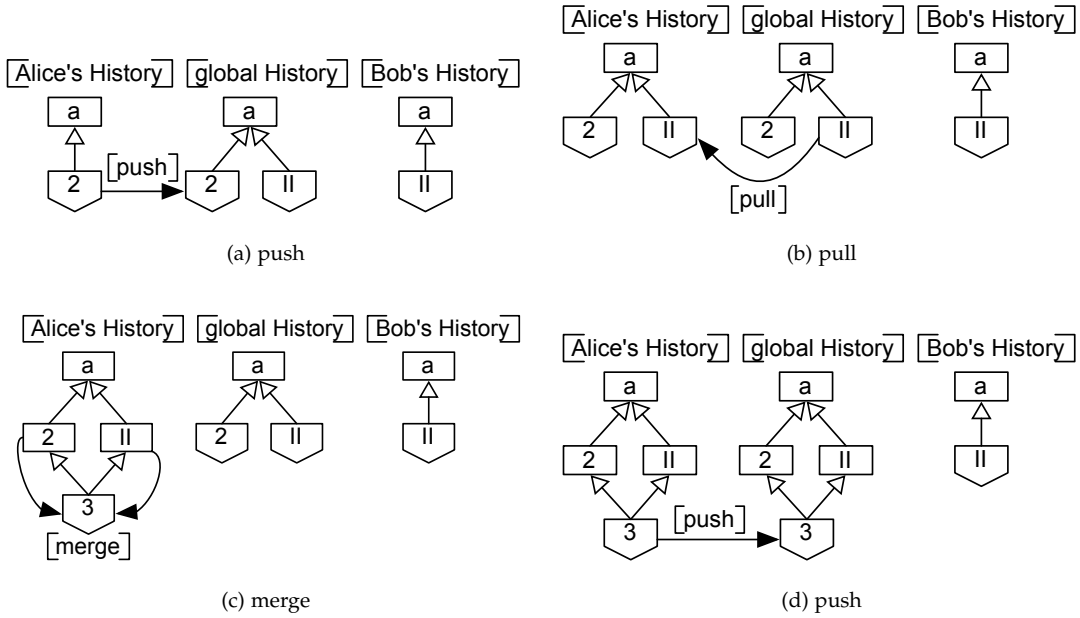
Although not covered in this example, a user could create a new module (see operation *O-3: Initialize module*), to start a new project, or clone an existing module (see operation *O-1: Clone remote module*), to start working on an existing project as well.

2. The developer modifies a number of files in several steps. Using the local commit operation changes are recorded as frequently as possible, creating a fine granular version history in the ***local module***, without sharing them with other users, as done by Bob in Figure 23b. N.B. Alice committed her changes before. They were not shared with Bob (see Figure 23a). If some modifications lead to unexpected results, it is often easier to reverse the changes and continue from an earlier snapshot. Using PlatinVC, any of the recorded steps can be reversed without network connection.

3. Once a planned functionality is implemented (or a bug is solved) and the project is in a stable state, the user can share all his locally recorded snapshots with all other participants, mimicking the behavior of a cVCS, by executing the push operation (see Figure 23c and operation *O-9: Push globally*), like Bob does in Figure 23c. This copies the stored snapshots form the local module to the ***Peer-to-Peer Module***, from where the snapshots are pushed to the virtual global repository once a network connection becomes available. To push the snapshots to the virtual global repository the developer's computer contacts specific peers and sends them the new snapshots, which integrate them in their peer-to-peer module. Thereby every user querying for the latest changes can retrieve them.

7.2.3 *Conflict Resolution*

In the previous described workflow conflicts might occur, if another developer shares his snapshots concurrently. The following example, illustrated in Figure 24, explains this situation. We start from the situation presented in Figure 23, where Alice and Bob committed changes made to the common base snapshot 1. Bob already shared his snapshot using the push operation. Alice does the same in Figure 24a. This leads to a conflict, as both snapshots are based on the same snapshot, and might contain changes that contradict each other. As usual



(a) push (b) pull

(c) merge (d) push

Figure 24: Conflicts and Resolution in PlatinVC

in dVCSs the snapshot is applied, but forms a second head in the (named) branch we look at. N.B. If Alice had pulled the latest snapshots first, as recommended and presented in Figure 16, this second head would be created in her local history only. Even if this procedure is followed, the presented situation could occur, as shortly after Alice pulled the latest snapshots, but before Alice pushes her local repository, Bob could push new snapshots, which results in a conflict as soon as Alice pushes her snapshots.

As we further see, the resulting history forms a direct acyclic graph (DAG). The two parallel ***development lines*** are referred to as **unnamed branches**, where both snapshots *2* and *II* are the latest head versions. In order to apply this history, one must choose the head to update his working copy to.

Through an acknowledgment message Alice is informed that an unnamed branch was created, and is asked to solve it. Alice reacts by pulling Bob's conflicting snapshot in Figure 24b. She merges the conflicting snapshots locally (see Figure 24c) and shares the resulting snapshot *3* as presented in Figure 24d. Now the two ***development lines*** are reunited, so that there is only one head again, making it clear where further changes should be built on.

If Alice does not receive the acknowledgement message due to message loss or being disconnected she invokes the push again. As it was already applied, the only action the system takes is to resend the acknowledgement message. If a conflict with another snapshot should arise, an unnamed branch would be created. A warning is included in this acknowledgment message. For each repeated push execution a new acknowledgement message is sent - so the message will eventually be received by Alice. Thus the author of a conflicting snapshot that lead to the unnamed branch, is informed about the branch with a high possibility and can take action to resolve the branch. The other author is not informed, as he could be tempted to also merge the snapshots, which might result in a new conflict, if multiple merged snapshots are pushed again.

The global history in Figure 24a has two ***head versions***, which are both the valid latest version in the respective branch. To avoid this confusing state, each push operation should be preceded by a pull request. The conflicting head versions could be merged locally first and published to the resulting DAG building the global history in Figure 24d. However, doing so is not always possible, e.g., when ***network partitions*** happen, as discussed in Section 8.6.

N.B. The server on a cVCS would prohibit Alice to share her changes. Alice would be forced to obtain Bob's snapshot first, merge conflicts locally, and commit only the new created snapshot to the server. The linear history would consist of the snapshots *a-II-3*, Alice's modifications which lead to snapshot *2* would not be recorded in a cVCS.

7.3 FEATURES

As mentioned before PlatinVC offers all features offered by a distributed version control system, namely mercurial [O'S09]. In this section we describe the additional features enabled by PlatinVC, which are novel to version control systems.

7.3.1 *Automatic isolation of concurrent work*

When a project is developed concurrently it is difficult to manage, which contributions should be applied immediately and which should be postponed to a later time, when the developed product is integrated. A single developer can keep his modifications locally by not sharing them with the push operation. But separating the work of multiple developers is challenging and, to date, no satisfying solution has been found. The classical approach coming from the centralized version control systems is to split the ***development line*** into branches. Teams share their changes in the created branches, which are merged into the main development line once all tasks are completed. Before branches are merged, no updates from other teams are received, being helpful (like a bug fix) or disturbing (like a half implemented function). Merging the branches is difficult, especially if there are a lot of changes in the artifacts of a branch. Conflicting modifications, which might have been created long time ago must be resolved. A major design goal of the ***distributed version control systems***, especially *Git* (see Section 5.2.3), was to handle branches in a more convenient way. The development of each developer occurs in an individual branch, which is merged whenever they share their changes. These individual branches are merged often and early. More importantly, they are merged by the persons who created them and are thus the most eligible to resolve possible conflicts. Nevertheless branches have to be merged, and due to the lack of a global repository they have to be merged multiple times. Let us assume that Alice and Bob exchange their changes by merging their branches, as presented in the previous example. Now Bob's local repository includes Alice's changes as well. If he shares his changes with Cliff, a version created by Alice, which did not conflict with Bob's versions, may conflict with one of Cliff's creations. Even worse, if Dave shared his changes with Cliff, and subsequently shares them with Alice, the same conflict between Alice's and Cliff's version will occur. Alice and Cliff might resolve them in a different way, meaning that Alice and Bob, and Cliff and Dave, will need to synchronize their repositories again.

PlatinVC aims to solve the problem of merging branches by avoiding their creation in the first place. Two mechanisms enable a workflow, where branches are almost not needed: Isolated folders and postponed branches. Both are presented in the following.

Isolated folders

All projects are organized using a folder hierarchy. Beginning with the outermost folder several folders exist, which separate a project into parts. In a typical software project, all files belonging to a specific function are included in the hierarchy of a folder. Only cross-cutting concerns [Par72] cannot be organized in this way, but normally they are not implemented by a single team of developers. A wide spread example for a cross-cutting concern are security issues, which often have to be cared about in various code places. However, a good design and/or using weaving techniques like aspect oriented programming can capsulate cross-cutting concerns in one central place (i.e., under one encapsulating folder).

PlatinVC utilizes this folder hierarchy to distinguish which updates to certain files are needed, and which would be ignored better. A developer can choose to update the files of a

folder and all subfolders only. He can also choose to update individual folders, without the need to update all files of their subfolders. In this way, only changes to the files included in the chosen folders are received; changes to other files made by different developers are ignored as long as the same developer did not change a file in the selected folders. N.B. While centralized version control systems offer the ability to update only specified folders as well (which is not possible in a dVCS), PlatinVC additionally retrieves related changes to artifacts in unspecified locations. Section 7.4.2 details how this update operation works. The idea behind this concept should be clarified by the following example. Let us assume that a software project, where Alice and Bob are working on, consists of one core part and two features, which are separated into three different folders. Alice's team works on one feature. Changes are pushed globally, but updates are only pulled from the folders of the feature they work on. In doing so, they automatically ignore all changes shared by Bob's team, who are working on the files residing in the other feature's folder only.

Specifying the folders for which updates should be exclusively received effectively separates the development on different parts of a project. If changes are needed, they will be retrieved automatically. If, for example, Bob's team finds a bug in the core folder of the project, which involves changing some files in the feature folder Alice's team is working on, Alice's team will receive these updates automatically. As all made changes of Bob's team *might* be important, all changes in all three folders are received. Alice's team can now decide, whether they want to merge the retrieved changes of Bob's team fully or partially (see ***cherry picking***) into their development line. If they decide to ignore the changes a postponed branch gets created, like explained in the next section.

A developer should only update the folder he is working on. The received updates are guaranteed to include all changes to files included in these folders, but also all changed files in other folders upon which the received updates were based on. N.B. The selective updates of the ***centralized version control systems*** do only retrieve the chosen artifacts blindly, without related changes that might be of interest. Most of the time concurrent work will be isolated. If unwanted changes are received, they can be ignored, which will create a postponed branch, to be explained in the next section. This section also explains how the development of multiple teams, which work on overlapping folders, can be managed.

Postponing Conflict Resolution

As presented in the example in Section 7.2.3, unnamed branches are created automatically when different developers base their changes to the same snapshot, but these unnamed branches should be merged before they are shared. Named branches exist as well. They should be used for long term separation of parallel versions, such as for maintenance, feature or variant ***development lines***.

The work of a single developer is isolated by committing it locally only. When a group of developers wants to share snapshots among themselves, a mixture of named and unnamed branches can be used. Every team member would work on his tasks and commit his fine grained modifications locally. A team member shares these local snapshots to the named branch. Unnamed branches, which reside within the named branch are created and merged. As we described in Section 7.3.1 the natural separation of a project in subfolders can be exploited. When only members of a certain team work on artifacts which are all in the hierarchy starting from a subfolder the automatic isolation of folders separates the teams snapshots from other developers without the need of a named branch.

The creation of unnamed branches could be avoided to a specific degree: As we will see in Chapter 8 in the first phase of the push operation a list of existing snapshots (only their ***metadata***) is retrieved. Based on this list conflicts can be detected by the pushing peer and the push process could be aborted, forcing the user to pull and merge the conflicting snapshots first. However, we did not implemented this behavior, as it cannot work reliable: In case of a ***network partition*** conflicts may remain unnoticed. Even under normal operation it might happen that a conflicting snapshot is submitted by another developer after the metadata was received, but before his own snapshots were sent.

All changes ever made are always retrievable, while the latest snapshot is a single head, which represents the project in a stable state. This behavior enables a workflow, where a developer does not need to plan branches in advance.

As explained in Figure 24 Alice and Bob can work on different features in the same branch. They commit their changes locally and push them globally, if the software project remains compilable. Before pushing local snapshots globally, Alice pulls the latest snapshot and merges it with the latest local snapshot in her working copy. If everything continues to work as expected, she also pushes this merged snapshot; if not, she discards the merged snapshot and pushes her local snapshots only. Doing so results in a diverging ***development line***, an unnamed branch as shown in Figure 24a. From now on, all her snapshots will continue to be added in her unnamed branch, while Bob will continue to work in his branch - without the need to be notified. Once their work is done they can merge the latest snapshot to one, recreating the single development line.

Due to the **automatically created unnamed branches**, solving conflicts can be postponed to the moment when it becomes inevitably, which is not possible in cVCSs, where conflicts have to be solved before changes can be committed. Postponed conflict resolution has to be planed, as branches have to be created before changes can be committed.

The handling of branches is inherited from the ***distributed version control systems***. Unlike centralized systems they bring the drawback that the work of multiple developers is distributed and has to be collected in a manual process, by exchanging snapshots which each developer individually. PlatinVC unites the immediate retrievability of all shared snapshots present in centralized version control systems with the ability to postpone conflict resolution spontaneously from the distributed approaches.

Unique to PlatinVC is the possibility of groups using unnamed branches spontaneously. A team can start working on a single branch. A group can split, if their development begins to affect one another, by simply not merging concurrent snapshots. As with the development in any other branch conflicts split up a branch into new branches. But when visualizing the history it is clearly visible which branch started as a sub-branch from a team branch. Nevertheless, by executing the pull operation prior to pushing changes unexpected branches can be avoided.

As in any dVCS featuring unnamed branches, they can be named at any later time. If snapshots are merged, their last common snapshot is identified and used in a three-way-merge. In contrast, in a cVCS the snapshot, where the development line was branched would be used, regardless of whether at a later time a more recent snapshot had already been merged with the original development line.

Merging Branches

It seems that the ability to create branches fast and to postpone conflict resolution, which is inherited from distributed version control systems, leads to many emerging branches, which are hard to merge. But practice reports from projects using distributed version control systems prove the opposite [BRB+09, Loe09b]. Branches are frequently created for experimental development. Just after a few snapshots they are discontinued or merged in to a main branch. It is easy to start a branch from any snapshot version, so a branch is merged back, when a single task is finished. Most of those branches are shared with other developers when they are already merged back into the main ***development line***. However, in distributed version control systems branches have to be merged multiple times, once for each developer pair, when they exchange their repositories. Moreover, concurrent work might remain unnoticed. Same features or bug fixes might be performed with slightly differing implementations. Resolving conflicts, or even worse noticing duplicated functionality, is challenging under those circumstances.

In PlatinVC branches can be created as easily as in distributed version control systems and merged only a single time, similar to centralized version control systems. We will see later that not always all existing snapshots might be retrieved. However, it can happen that due to network problems some snapshots in a branch are not retrieved. This can affect the latest snapshots only, as PlatinVC provides ***causal consistency***. All base versions of the latest

retrieved snapshot are retrieved as well. If a branch is merged on this basis only the missing snapshots have to be merged, once they are retrieved. Merging those remaining snapshots is easier, as all previous snapshots are already merged and conflicts among them are resolved. Effectively merging the missing snapshots is equally to merging a complete branch with the same snapshots.

7.3.2 *Working Offline*

Even if a user has no network connectivity, he can use PlatinVC to track his changes. This is especially helpful for ***nomadic developers*** who are able to work while traveling. As with any dVCS, local changes can be tracked by committing them locally only, to the ***local module***. A global push can be executed being offline, which marks the locally recorded snapshots to be pushed automatically once there is a network connectivity. Actually the push operation pushes the recoded snapshots from the local module to the peer-to-peer module. As soon as the peer rejoins the ***overlay network*** its peer-to-peer module is synchronized with the peer-to-peer module of other peers (i.e., the virtual global repository), which shares the offline created snapshots automatically.

In this process already existing snapshots are not transferred again. Offline created snapshots could be already shared in the overlay network by a different developer, with whom these snapshots were exchanged before, e.g., using the underlying dVCS's capacities (like copying patches over USB flash memory drives). This other developer may have already shared all snapshots with the other peers in the overlay network. If the other developer recorded additional local snapshots, only those snapshots not yet shared are pushed to the virtual global repository.

One limitation, however, is that conflicts will be noticed delayed, when they are actually being exchanged with other machines. As detailed in Section 7.2.3 conflicts do not hinder the application of a snapshot but form an ***unnamed branch*** instead. This will be noticed delayed at the time the networks, where the contradicting snapshots were applied to, rejoin. It is sufficient to merge only the latest snapshots of the diverging branches.

7.3.3 *Interoperability*

PlatinVC inherits the ability to interoperate with other version control systems. The history in the local module can be exchanged with a number of other systems, in any combination. As explained in Chapter 8, the local module is a full blown dVCS. In Section 5.2 we mentioned the implementation of several adapters, which can exchange snapshots with other systems, continuously through synchronizing or in a sequential batch process. Nearly all dVCS are able to synchronize their changes in an interoperable manner. The most popular cVCSs can also be connected. However, as their history can store just a subset of the history in a local module of PlatinVC, they are not identical copies. Depending on the adapter and workflow used unnamed branches are, for example, either ignored or mapped to named branches.

Once the local module is synchronized with an external VCS, the introduced snapshots can be pushed to the corresponding global module, making them accessible to the other participants.

7.3.4 *Offers all dVCS Operations*

As detailed in Chapter 8, the local module of PlatinVC is a dVCS repository, which is accessible by the user. We utilize this dVCS for basic VCS operations. This enables the user to use the underlying dVCS with all its operations, such as ***bisect*** and similar helpful features. The local module can even be exchanged with other developers just using the dVCS. However, using the operations provided by PlatinVC is advised, as they take care about with whom to share and from whom to get new snapshots.

It should be avoided to change parts of the history, which were already shared with the virtual global repository in PlatinVC. This includes ***rebasing*** the history or removing snapshots. Doing so would create new snapshots, although only their identifier is changed and the content (the ***diff***) remains the same. The identifier would change because the moved snapshot is based on another snapshot, whose ID is included in the moved snapshot. As we explained before the snapshot's ID is based on its content, so changing a basis snapshot changes the snapshot's ID. If the history in the local module is changed after it was pushed in PlatinVC, and is pushed again, the resulting history in the global module would combine the moved snapshots in their old and new location, certainly confusing all participants. Even in a pure dVCS this behavior should be avoided, as the result would be the same (if we substitute the virtual global repository of PlatinVC with another user's dVCS repository).

7.3.5 *Backing up Artifacts Redundant*

Multiple copies of the repository are stored by multiple machines. The risk of losing any stored artifacts is very limited. A user stores all of his changes in a local repository on his machine. If this storage were to be corrupted the user could restore the complete history of all artifacts she created by retrieving them from the system. There are a minimum of x copies of any version of any file, where x is the number of ***replicating peers*** +1 for the maintaining peer. The replicating factor x is configurable. A higher number ensures more failure resistance and availability, whereas a low number reduces the time for an update. The value of x is at least 2, but typical 4. The replicas are randomly chosen (more precisely: their are chosen by their identifier, which is randomly assigned). Thus it is unlikely that they are part of the same subnetwork. But even in the unlikely case that all replicating peers and the maintaining peer are failing before they can copy the latest updates on a substituting machine, there is a chance that all versions are still retrievable. Whenever artifacts are committed in a snapshot, all maintainer (and their replicating peers) store the complete history of all artifacts in the snapshot, regardless of whether they are responsible for them. The history of all artifacts in a snapshot contains all snapshots the current snapshot is based on, and also their base snapshots, which might additionally contain versions of artifacts other than the ones included in the latest snapshot. If all of these copies are unavailable, any peer who updated its local repository can provide some versions as well. When failing peers rejoin, they offer formerly stored versions again.

Due to the highly redundant storage of committed versions in the system, the total loss of an artifact is very unlikely. Every storing peer would have to lose his local storage permanently. The older a specific version is, the more likely it is that many peers have stored it. The more artifacts of different folders are modified by a single developer in a snapshot, the more maintainers store that snapshot. Only very recently committed snapshots, or snapshots consisting of artifacts developed in isolation (changed by developers, who did not change artifacts from other folders afterwards) are endangered to get lost - which only occurs if their maintaining peer, along with his replicating peers and the authors of the endangered snapshots fail simultaneously, without enough time to update a substituting peer. And only if the physical storage of these machines is destroyed are the snapshots lost permanently.

7.3.6 *Degree of Consistency*

As detailed in Section 7.4 PlatinVC offers a number of different ways to obtain updates from the virtual global repository. In Section 4.4.2, different degrees of consistency were discussed. With regard to single artifacts, ***sequential consistency*** is guaranteed, as all versions are stored on a distinguished machine. In order to retrieve the latest updates of a set of artifacts recorded in a ***snapshot***, a number of machines have to be asked. When the network is undisturbed by ***failing peers*** or lost messages, ***sequential consistency*** is archived. However, in a realistic network these factors cannot be ignored, therefore in the worst case only ***causal consistency*** is guaranteed, as the latest snapshots stored on some unavailable machines cannot be retrieved.

If a snapshot is retrieved all ***basis*** snapshots are retrieved as well, as they are stored on the same machine the snapshot has been retrieved from.

7.3.7 *Support for Traceability Links*

Cross References

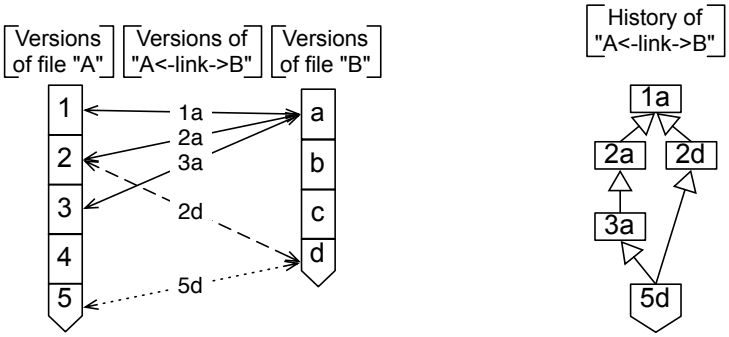
Links can be created between any items, be it an artifact, folder or module, in any version. A link itself is a version controlled item. A link is bidirectional and connects two artifacts or folders in a specific version in a specific branch. It can store any metadata about the connection of the linked items, such as integrity constraints, which can be evaluated to prove whether a connection is justified. This validation, however, is not handled by PlatinVC. The system PlatinVC does not control how links are interpreted, it merely tracks their concurrent evolution. This enables different, mostly locally operating tools to use and manipulate this information for multiple purposes.

The links can be used to store dependency relations among the items, which enables configuration management to choose specific versions. For example, a project could link to another project, indicating the minimal needed version it depends on. Individual artifacts can be linked to store traceability [WN94] information. Hereby it is possible to connect artifacts which are created during a development process. E.g., a requirements document could be linked to a design document, which is linked to the implementing source code. This source code could be linked to a test case, storing the result of the test in the link as well. If the budget for a project is short, only critical requirements could be tested by tracing the connection from the requirements document to the test case. By storing integrity constraints, the connection between two items can be validated. Updates on the content of linked artifacts could harm these constraints, which could be marked in an update on the link. A supporting tool could enforce further changes to either file to repair the broken connection.

Once a link is created it cannot point to a different item. If a link should point to a different goal, the old link document is marked as deleted and a new document is created, which connects the new items. Similar to a file rename explained in Section 5.2.4 changes to the old document follow the new document.

Concurrent Updates on Linked Artifacts

Changing the stored information updates a link document to a new version. Pointing to a different version of a linked artifact changes the link document. As a convention a link should only point to more recent versions of the linked artifacts in an update. Figure 25a illustrates an



(a) Files and versions

(b) Created history of the link document

Figure 25: Concurrent updates on a linkdocument

exemplary situation. The left column represents five versions of artifact “A”, the right column four versions of a linked artifact “B”, and the arrows in between represents the versions of the linking document (representing a binary traceability link). The version of both artifacts are created by different users. Alice works on “A” and Bob on “B”. First Alice creates the artifact “A” and links it to version a of artifact “B”. She updates “A” to version 2, updates the link to version 2a, and repeats the same process later to create version 3 of artifact “A” and version 3a of the link document. The versions 1a, 2a, and 3a of the link document are represented with the solid arrow lines. Meanwhile Bob updates artifact “B” to the versions b, c, and d. He updates the link to connect version d of artifact “B” and version 2 of artifact “A”, which is the latest version he fetched.

Alice created the version 3 of artifact “A" and updated the link from version 2a to version 3a at the same time when Bob created the versions b, c and d of the artifact “B" and updated the link form version 2a to 2d. Hereby a conflict in the traceability link’s version history arose. This conflicting history is stored by PlatinVC in an unnamed branch (as illustrated in Figure 25b). Version 3a and 2d are both based on version 2a. This branch can be merged by pointing to a more recent version of the connected artifacts. Thereby link version 5d can be based on 3a and 2d to merge the diverging history lines. To avoid unnamed branches, updated links should always point to the latest version of the linked artifacts. When the network is separated into ***partitions*** (see Section 8.6) this is not always possible immediately.

7.4 SERVICES

In this section the services offered to the user by PlatinVC are detailed. Chapter 8 and Chapter 9 explain how they were realized. In addition to PlatinVC’s services, all commands of Mercurial (see Section 5.2.4) can be called, which manipulate the local module only. To share resulting changes with other developer the following services of PlatinVC are to be used.The *module operations* connect a module to PlatinVC, the *retrieve* and *share* operations exchange recorded snapshots among the users.

7.4.1 *Modul Management Operations*

The module operations manipulate the modules of PlatinVC, namely the local module, the corresponding peer-to-peer module and the respective global module in the virtual global repository (as detailed in Section 7.1). Only the local module is visible to the user, the other modules are created or updated during the execution of these operations by PlatinVC. The following operations are to be used to connect a module to PlatinVC, which will serve as the local module.

O-1: Clone remote module: A module can be accessed by cloning it from the ***global repository***, by addressing it with its unique module identifier. The entire module, which contains the complete history of the contained project, is transferred using PlatinVC over the network in this way. Although the module is rarely larger than double the total size of all artifacts in the latest version, it may still be several mega bytes. Due to the typical network speed, it may be better to share a module with a new participant by handing him a copy on a data medium such as an USB flash memory drive. Copying data to or from a medium is faster than typical network transfer speed. The bandwidth in the network could be put to better use in other tasks. If some participating developers are physically close to each other, obtaining an initial copy of a module via the network connection should be avoided.

O-2: Add an existing module: This service adds a module that already exists in the local file system, to be shared with other developers by PlatinVC. Modules can be large, wherefore it may be more efficient to copy them by means other than over a network connection. The specified module is used as a local module by PlatinVC. It is synchronized with the corresponding global module - if it does not exist yet, it will be created. The peer-to-peer

module, which serves as an intermediate connection between the local and the global module, is created in this process as well. After this operation is finished the added module can be used with PlatinVC.

O-3: Initialize module: A new module is initialized, by creating all necessary local and global components (i.e., the local module, peer-to-peer module and the global module in the virtual global repository). The user is warned, if a module with the specified name already exits, and offered to clone the remote module instead.

O-4: Remove module: Removes a local module from the managed modules. The corresponding peer-to-peer module is deleted. The local module itself remains undeleted (the user can erase it, if desired). The corresponding global module in the virtual global repository is not deleted either. As long as any participant has a local copy of this module she might continue to work with it. As a client might be not connected to the network it is impossible to see, if the last user having the module removes it. This is not a constraint of the used technique but merely a design decision. If desired a simple mechanism[1] could enable to remove a module completely.

7.4.2 *Retrieve Operations*

The retrieve operations collect the latest snapshots shared by other users.

In the most centralized version control systems, updates are retrieved for all artifacts in the module, but from a selected branch only. Updates in a distributed version control system are retrieved in two steps: first all snapshots in all branches are added to the local history, then the user applies a chosen snapshot from one branch to his working copy.

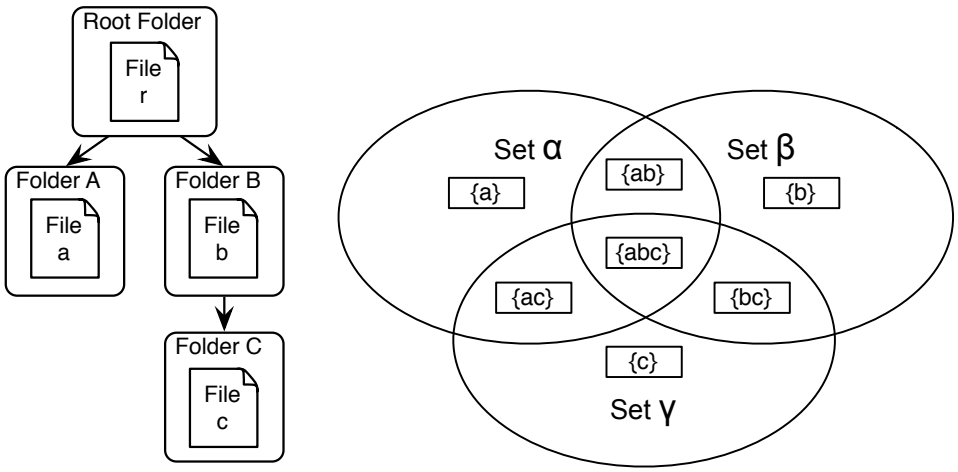
In PlatinVC all retrieved snapshots from all branches are applied to the local history in the local module. With the commands of the underlying dVCS, a user can subsequently choose which snapshot of which branch he wants to apply to his working copy. A user can optionally chose to update all folders starting from a specified folder, instead of all existing folders in the working copy, as explained in Section 7.3.1. As detailed in Section 8.4 only one peer has to be contacted to check for updated artifacts for each folder. Specifying a folder other than the outermost folder distributes the load away from the outermost folder's maintainer.

The following example, illustrated by Figure 26, clarifies which snapshots are retrieved: Let us assume our project consists of the artifacts and folders presented in Figure 26a. Each folder has one artifact, represented by a file. The topmost folder of the project is the root folder, which contains the folders *A* and *B*. Inside folder *B* is the folder *C*. If we track modifications on the files and commit them seven different snapshots could be created, depending on which files we modified. If we modified the files *a* from folder *A* and *c* from folder *C* the resulting snapshot would be *ac*. Figure 26b shows all possible combinations, ordered by sets. Figure 27 shows what the history could look like, when modifications on different files were committed. A box represents a snapshot. A modified file results in a new version, indicated as filename+roman number (which is the sequential version number) in the figure. The superscript number indicates a variant of the version. A snapshot is indicated by the set formed by its containing versions.

The initial snapshot contains the first version of the files *a*, *b* and *c*. Based on this snapshot seven different developers modified a different combination of files. The first developer changed the content of the three files *a*, *b* and *c*, which resulted in the snapshot { aII^1; bII^1; cII^1}. Another developer changed the files *a* and *b*, which resulted in the variant snapshot (an ***unnamed branch***) { aII^2; bII^2}, and so on. Afterwards all snapshots were merged[2]. In the

1 (1.) Keep track of users, remove a module, if the last user executes the remove operation. This would introduce additional messages to keep track of the number of users.
(2.) Remove the global module immediately, meaning deleting all local and peer-to-peer modules on all storing peers, once they are online. This requires a trusted environment to work, where a remotely executed remove is locally enforced, and not available to everyone.

2 For simplicity this is presented as if it occurred in one step. In PlatinVC only two snapshots can be merged, which would require several intermediate snapshots.



(a) File hierarchy (b) Sets of Snapshots

Figure 26: Example to clarify which snapshots are pulled.

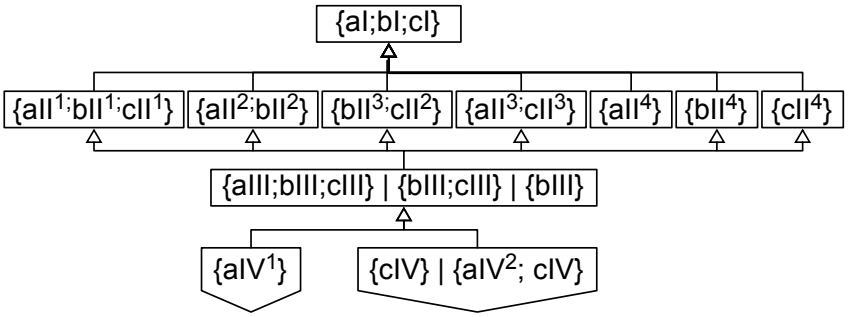


Figure 27: Exemplary history (simplified)

merged snapshot either all files were changed, leading to new versions and forming snapshot { aIII; bIII; cIII }, the files *b* and *c* were changed (forming { bIII; cIII }) or only the file *b* was changed (resulting in { bIII }). We introduced three alternatives here, which are used in the example. Based on this snapshot the developers changed the file *a*, thus created snapshot { aIV^1 } and the file *c* ({ cIV }) or *a* and *c* ({ aIV^2; cIV }).

We assume that a further developer has the initial snapshot only. He has several options to update his local repository. He executes *pull using specified folder*. He works on a part of the project, which is represented by files stored in the folder *B* or any subfolder under *B*, namely the files *b* and *c*. He is only interested in modifications, made while working on this part of the project. If he pulls the latest snapshots specifying folder *B* he will get all snapshots, which are in the united set β and γ, as presented by Figure 26b. That would retrieve all snapshots presented in Figure 27, except the snapshot { aIV^1 }, as the latter is in the disjunct set α. All snapshots in the set β (and its intersections with the sets α and γ) are guaranteed to be retrieved. The snapshot { cIV } or { aIV^2; cIV } respectively, which are in the disjunct set γ could also be missing, depending on the message loss in the network. Only if the folder *C* was specified using the pull operation, the latter snapshot would be guaranteed retrieved as well. N.B. All snapshots leading to the snapshots { aIII; bIII; cIII }, { bIII; cIII } or { bIII } are guaranteed to be retrieved as well, although some are not in the united set β and γ (like { aII^4 }).

As detailed in Chapter 8, the repository is distributed by folders instead of snapshots. This decision is reasoned by the fact that software projects are organized in a hierarchical folder structure. In a good design, different aspects of the software are separated into different

folders. It seems more natural to work in different folders instead of different (named) branches, especially when updates can be received in isolation, as presented in Section 7.3.1. PlatinVC was built to reflect this point of view, as detailed in Chapter 8.

O-5: Pull globally using specified folders: Gets all locally missing snapshots for a folder and its subfolders, which correspond to a branch of the tree representing the folder hierarchy of the project. As illustrated by the previous example and explained using Figure 26b, all snapshots in the disjunct set α would not be retrieved. The snapshots in the disjunct set γ can be missing, if messages are lost, and the snapshots in the set β and its intersections to the other sets are retrieved guaranteed. All former snapshots, the latter is based on are also retrieved, if not already present.

O-6: Pull globally: Acquires locally absent snapshots of all artifacts and applies them to the local module. This is equivalent to executing operation *O-5* by specifying the outermost folder of the project. As explained before, not all snapshots might be retrieved under guarantee. If all existing snapshots must to be retrieved, a variant of this operation could be called, which executes operation *O-5* specifying all existing folders. This operation would involve many machines and be inefficient, and should only be used in situations, where the operation duration is less important than the completeness of the retrieved snapshots. Packing a software project to form a release could be such a situation.

O-7: Get Artifact: This operation retrieves the latest version of a single artifact. Unlike the other operations, the history is not updated and the file is transferred in its full form. This is useful if a user does not want to use the version control system, but needs to retrieve the stored artifacts. This is the case in a wiki engine, when articles are accessed for reading.

7.4.3 *Share Operations*

Committing changes is a twofold process: first changes are **committed locally**, which records the modifications made to all files in the ***working copy*** in a snapshot. Subsequently snapshots can be **pushed globally**, which publishes all unpublished snapshots to be accessible by all participants.

By splitting up the commit operation into two steps, the commit frequency recommended in Section 4.2 can be implemented: Users can commit any changes in short intervals locally only, benefitting from fine-grained control over the evolution of the project. Thus small changes can be undone and variant development tried out, without concern about destabilizing the project, i.e., temporarily breaking the ability to compile a software project. When the project is stable again and does not hinder coworkers in working on their changes, all local committed snapshots are pushed globally in one step. Conflicts are detected and the solving mechanism takes over.

As detailed in Section 7.3.2, all formerly not exchanged snapshots are pushed as soon as a network connection is available.

O-8: Commit: Commits changes to the local module. Each artifact that had its content modified after the last commit is recorded as a new version. All versions formed are part of a newly created snapshot. This snapshot is not shared until push is initiated. This operation calls the underlying dVCS's commit command.

O-9: Push globally: All snapshots not yet pushed are applied to the virtual global repository. After the operation finishes, any participant can retrieve those shared snapshots.

O-10: Commit globally: Commit globally mimics the committing operation of cVCSs and is realized by committing changes locally and subsequently pushing them in one step.

7.5 SUMMARY

This chapter presented an overview of the architecture of PlatinVC. The system works without relying on any single participant, as is usually the case in dVCSs, while offering the centralized, up to date view of a cVCS. It combines the advantages of these systems without inheriting their shortcomings. PlatinVC combines the concepts of centralized and distributed version control to enable a novel workflow, not possible in either system. The locally operating features of a dVCS can be fully utilized by a user. PlatinVC inherit's the ability to work without network connection, automatic branches and interoperates with selected version control systems, while providing the ability to retrieve any shared version with the global view of a cVCS. PlatinVC extends these basic features with mechanisms that manage the evolution of traceability link and offer a novel approach to separate the work of developers. Other than using them for traceability of artifacts, a basic configuration management is enabled by a snapshot based version control and links between files, folders and modules, of which evolutionary changes are tracked. The automatic isolation of concurrent work enables a novel workflow, where users specify the artifacts they are interested in, which will retrieve related updates on other artifacts as well (unlike the selective updates in cVCSs). Branches are created automatically, when conflicts arise, and do not have to be created in advance. PlatinVC provides a solution to the problem of finding the right frequency to share changes by providing personal commits, which are shared globally with all other participants once developers finishes their task.

8 DESIGN OF PLATINVC

The final design of PlatinVC presented in this work is the result of five development iterations (compare to Appendix A).

8.1 DESIGN PRINCIPLES

Distributed programs have a higher complexity than their single machine counterparts. They run in a less predictable environment, which makes development and testing harder ([TS06, CDK05]). PlatinVC was developed based on the following design principles. They are derived from general software design patterns [GHJV95], the area of distributed systems ([CDK05]), design principles typical in peer-to-peer systems [AAG+05] and our own experience gathered while refining the several versions of PlatinVC, which resulted in the presented prototype.

D-1: Reuse existing technology: In order to increase the maturity of the prototype, existing solutions should be integrated whenever possible. Software developed in an educational institute often lacks the maturity commercial or even open source projects have. Normally there is lesser man power and time spent in the development process and the software is less intensively tested. Successful projects are at least partially developed as open source projects over multiple generations of doctoral students. Therefore, wherever possible, software, which proved to be mature, should be reused. Preferably without changing the software or by participating in its development process, so that new releases can be integrated with minimal adaption to one's software.

D-2: Do not rely on the availability of any machine: A peer-to-peer network typically consists of unreliable machines. Any user can leave the network at any time. One reason for this behavior is that a client's machine is typically not as fail-save as a modern server system, thus they can fail even when unintended by the user. If a particular machine is essential to the system, its user would not be free to turn it off at any time. Therefore a failing machine should be considered as the normal case rather an exception (compare to [AH02]).

D-3: Do not rely on any message transfer: A peer-to-peer overlay network uses TCP/IP or UDP/IP messages, which are sent over unreliable networks such as the Internet. Therefore message loss should be considered frequently and countered with a minimal handshake protocol, where the receiver acknowledges a message which is otherwise resent after a timeout. To comply with design principle *D-4: Avoid transactions* this handshake should be avoided wherever possible (e.g. if receiving the message is not crucial for the system).

D-4: Avoid transactions: [FLP85] pointed out that it requires additional synchronization among distributed machines to agree on a single value, i.e., if a transaction can be completed or not. The unreliable nature of a peer-to-peer network, where any machine can fail or its messages can be lost, intensifies this problem even more. Basically messages and decisions have to be acknowledged among the deciding peers, failing peers have to be replaced in this process, and timeouts have to abort without terminating processes. Often otherwise acceptable transactions are aborted as a result of network disturbances.

The paxos algorithm, presented in [Lam01] and [Lam98], promises to be a practical solution. It was altered to the paxos commit protocol in [GL02] and adapted to a peer-to-peer network by [MH07] and [SSR08]. We compared these solutions with our own approach in [Voc09], which proved to be superior for transactions with approximately ten items by significantly reducing the time required to complete a transaction.

Nevertheless, all solutions bring an unavoidable overhead in the number of messages and the operation time of the commit protocol. With rising network instability, for example, emerging from leaving and joining peers, a transaction protocol is more likely to abort and has to be repeated, multiplying the overall costs. It is best to avoid any kind of transactions whenever possible.

D-5: Avoid storing statuses: A proven mechanism to provide availability in an unreliable peer-to-peer network is to replicate all stored information. The ***routing mechanism*** in a ***structured peer-to-peer overlay network*** takes care to deliver a message to the next closest peer once the original recipient fails. A number of neighbors of any peer, i.e., the closest peers according to their identifier, replicate all information needed to substitute that peer in case it leaves the network. Therefore all information must be copied to the neighbors, as soon as they are created or stored on the maintaining peer, in a reliable manner. A message has to be acknowledged with an answer or resent after a timeout. Again, this mechanism reduces the system's performance, as more messages are created and the time needed to store information reliably on a peer is prolonged by the time needed to replicate it successfully on the peer's neighbors.

Replicating information should be avoided by avoiding the creation of permanent information in the first place. Whenever possible storing a status should be avoided.

8.2 DESIGN OVERVIEW

To be compliant to requirement *R-14: Availability* PlatinVC is implemented in Java [Jav], enabling the system to run on multiple platforms.

PlatinVC consists of three main components, which are shown in Figure 28. These main

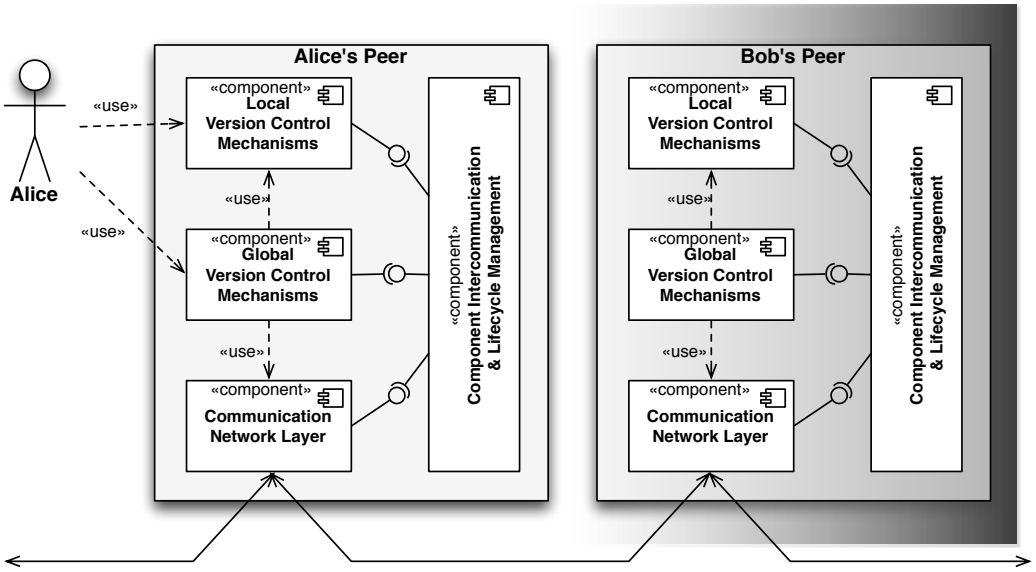


Figure 28: High level architecture of PlatinVC

components implement are mechanisms that enable a *global version control*. These mechanisms work as distributed algorithms, which are executed by a number of peers. The services of PlatinVC, which are described in Section 7.4, are triggered by a user. Additionally, some mechanisms react to failures in the system and repair faulty behavior.

These global version control mechanisms communicate among the participating machines using the *communication network layer*. This component realizes a ***structured peer-to-peer overlay network***, which routs messages using a virtual peer identifier and maintains the overlay network structure by replacing failing and incorporating joining peers.

The communication between the components is handled by the *component intercommunication and lifecycle managing* component, depicted on the right side in Figure 28. Using well-defined interfaces other components can use each other's services. A component could be exchanged with an updated version if needed.

A user ***commits*** changes to an artifact in her ***working copy*** using the *local version control mechanisms*. She shares the recorded snapshots with other users by executing the ***push*** operation of the *global version control mechanisms*. Likewise the latest versions of other users are ***pulled***, using our global version control mechanisms, and applied to the working directory by calling the ***update*** operation, which is part of the local version control mechanisms. For all core version control functionalities PlatinVC need to perform, such as calculating a missing ***delta*** that needs to be transferred, the services offered by the local version control mechanisms (provided by Mercurial) are used.

The reused technologies are briefly described in the next section, while the global version control mechanisms are detailed in the remainder of this chapter.

8.3 CHOICE OF THE SUPPORTING SYSTEMS

Following design principle *D-1: Reuse existing technology*, components which have proven to be reliable in various open source and industrial projects were utilized whenever possible. To the author's best knowledge each component represents one of the best solutions in its area.

8.3.1 *Component Intercommunication & Lifecycle Management*

To manage the communication among the components of PlatinVC, an implementation of OSGi [All07] was chosen. Basically this was needed to integrate PlatinVC with other tools, which use the same peer-to-peer communication instance. Using this technique ASKME (our communication application, presented in Section 9.3.4) has been easily integrated, as detailed in Chapter 9.

OSGi is a framework standard used in embedded systems where a small memory footprint and efficient, often real-time critical execution is important. We used the most popular implementation Equinox [equ], which is the core component of the Eclipse integrated development environment [Fou] since 2003. Equinox is, like Eclipse, an open source project, which is implemented by full time developers from global companies like IBM.

Equinox can handle a component's lifecycle, uninstalling old components and starting their updates without needing to shutdown the machine the software is running on (as demanded by requirement *R-20: Continuous expendability*).

The main benefit gained by using the Equinox OSGi framework is that PlatinVC could be combined with the widely used integrated development environment Eclipse, as well as running as a stand-alone application, using the same core logic with different user interfaces, as detailed in Chapter 9.

Alternative Choices

If the ability to integrate PlatinVC with other OSGi based tools is not needed, and requirement *R-20* can be left unsatisfied, any programming language's capacities to separate a program into components that interact using well-defined interfaces would have been sufficient.

Otherwise, there is no alternative as potent as an OSGi implementation. Among the few Java based implementations Equinox is the most extensively tested, as Eclipse is one of the most widely used tools in software development.

8.3.2 *Communication network layer*

As concluded in Chapter 6 an implementation of a ***structured peer-to-peer overlay network*** should be exploited. Pastry [RD01b] is one of the best performing ***peer-to-peer protocols***: it is

robust against high ***churn*** [RGRK04] and locates any peer in $O(\log N)$ routing hops, where N is the number of peers in the network.

The most promising Java based implementation is FreePastry [FP], which was designed by the original authors of Pastry [RD01b] in 2002 and is still under development. It is an open source project licensed under the very liberal BSD license. FreePastry is used successfully by numerous projects[1].

Alternative Choices

There are some peer-to-peer protocols, such as Kademlia [MM02], which show a better performance with regard to message transfer times and robustness. However, at the time of writing there is no mature Java based implementation.

Some of the developers of FreePastry started another Pastry implementation in 2003, which showed to be more reliable under heavy ***churn***. This implementation, MSPastry [CCR04], is developed by Microsoft as a closed source project. The project, however, is available to educational institutes. The user base is therefore smaller than FreePastry's user base. MSPastry is implemented in C#. The restrictive license agreement and the less widely available documentation render this alternative unattractive next to the prototype's appealing properties.

8.3.3 *Local version control mechanisms*

To implement the ***local commit*** and handle other local version control tasks (as required by requirement *R-1: Support common version control operations*) the version control logic of a distributed version control system seems to be tailor-made for our purposes. We chose to reuse an existing application: the ***distributed version control system*** (dVCS) Mercurial (see Section 5.2.4 for a detailed presentation). A dVCS brings the version control logic of a version control server application to a single user's computer in a lightweight and efficient way. Most operations are in fact faster than in a ***cVCS***, because data is transferred within a computer instead of over a slow network channel. It fulfills numerous requirements, such as requirement *R-8: Offline version control*, and it is fast in manipulating a module's history.

Mercurial runs on any platform which can execute python scripts. For performance reasons a very small number of helper components are written in c++, but are compiled for a majority of systems.

Alternative Choices

The discussion in Section 5.2 concluded that at the time of writing only two ***dVCS*** applications are in the same time feature rich as well as efficient: Mercurial and Git [HT]. Both systems differ only in some minor design details. The main operations, which fulfill all requirements listed under requirement *R-1: Support common version control operations* with a local scope only, are identical. The internal structure, in which snapshots are recorded, differs slightly.

Git's operations require slightly less time to complete and the repository size is slightly smaller in most cases as well [Git]. As a drawback all versions of a single artifact are stored in an indirectly connected structure: To find the previous version of a specific artifact Git first identifies the associated snapshot and traverses the parent snapshots until a snapshot is found where the artifact has been changed. Git achieves a small repository size by storing each version as a full version (not using ***delta compression***) and compressing all files subsequently. Thus, a repository, which is distributed into parts, which are stored among multiple peers, cannot be compressed as if all parts would be present on a single machine. Mercurial uses ***delta compression*** to store all versions of a single artifact, which could be better distributed among several peers. However, the main reason against Git is its platform dependability: Git was created for UNIX and Linux based machines and does not run on Windows without additional tools[2].

1 Some are listed in http://www.freepastry.org/projects.htm.
2 namely Cygwin (http://www.cygwyn.com)

8.4 GLOBAL VERSION CONTROL MECHANISMS

In this section we present the general mechanisms to share a developer's changes using PlatinVC. We first describe how recorded versions are stored among the peers in Section 8.4.1, Section 8.4.2, and Section 8.4.3. In Section 8.4.4 we present the mechanisms which are executed in order to fulfill a user's requests. How conflicts are handled is detailed in Section 8.4.7. Finally in Section 8.4.8 we present some additional mechanisms that handle renamed, moved, and new artifacts.

8.4.1 *Storage Components on each Peer*

All data in PlatinVC is stored distributed among the individual peers. Each peer acts as an active client as well as a passive server. So the actual versions of all artifacts are produced and stored on a peer, and replicated and offered by the same and other peers. The Metamodel in Figure 29 shows the structure of all stored data on a peer.

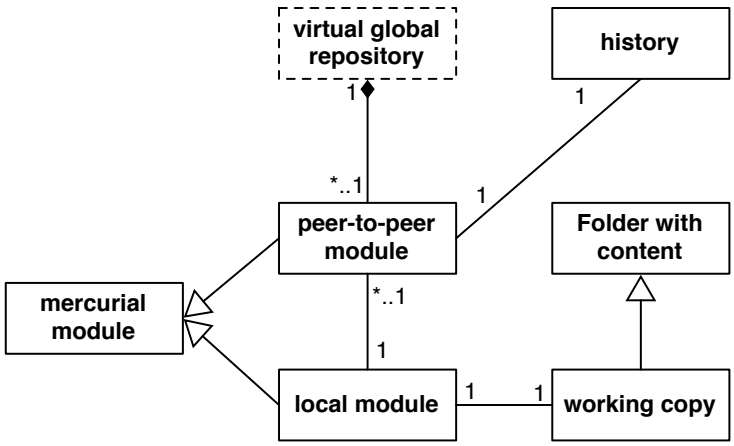


Figure 29: Metamodel of the components on a peer

Component for local Version Control

A user edits artifacts in a ***working copy***, which is represented by files in a folder hierarchy. The working copy is controlled by Mercurial, which is exploited as the local version control mechanism. A user creates snapshot by executing Mercurial's "hg commit" operation. The resulting history is stored in the **local module**, which is under the complete control of a user. Therefore all operations of Mercurial or any of its supporting tools can be used without network connection.

Besides committing snapshots locally, which are not yet intended to be seen by others, local temporary clones for experimental changes can be created, or a bug-free revision can be sought in the history with **bisect** (detailed further in [O'So9]), to name just a few operations. The complete history of a module can be accessed.

A user can use any of Mercurial's operations on the local module, and can even share it with other users using any of the workflows presented in Section 4.1. Nevertheless, whenever a local module's history should be shared with coworkers our system takes over. N.B. Mercurial's sharing mechanisms could be invoked by the user as well, but this should be avoided, as it brings the drawback of only being able to share one's changes with a limited number of other developers. PlatinVC automatically chooses the right peers, with whom snapshots are to be shared in order to make them publicly available, or in order to obtain the latest updates.

Component for global Version Control

All operations provided by the global version control mechanisms synchronize the local module with the respective **peer-to-peer module** on a user's machine. The peer-to-peer module represents the partitioned **virtual global module**, which is retrieved by combining the distributed parts from different peers, as detailed in Section 8.4.2. The *virtual global module* does not exist on any peer - it is only a conceptual construct, which represents the union of all *peer-to-peer modules* on all peers. The *peer-to-peer module* is in fact a second Mercurial module, but it is hidden from the user. Only the operations of PlatinVC should manipulate it. Initially a user ***clones*** a module from the global peer-to-peer repository. If the peer-to-peer module already exists it is updated when the user joins: First operation *O-6: Pull globally* is executed, which pulls all missing snapshots from a peer in the system which has the latest updates. Afterwards all snapshots shared using operation *O-9* while the user was offline are pushed to the maintainer of the changed artifacts. Only missing snapshots are transferred. If a user recorded snapshots which he shared with another user, using Mercurial's share possibilities, and that other user already integrated those snapshots in the global peer-to-peer repository the already present snapshots are not transferred again.

Each peer stores for each *peer-to-peer module* a **history cache**. Here the metadata of snapshots are stored. This **metadata** includes a snapshot's identifier, its parents identifier and the folders whose content was changed in a snapshot. All other information and the actual changes, which formed the snapshot are not stored. If the storing peer is responsible for a folder, which does not contain artifacts, of a module, which is not checked out by the user (and thus the peer does not store a corresponding local module), there is no corresponding peer-to-peer module stored. The entries in the history cache are identical to those in a corresponding peer-to-peer module (if present), but there are additional entries most of the time. As a side effect of the system's operations (share, retrieval and maintenance), the metadata of the snapshots in the distributed peer-to-peer modules are exchanged. A history cache thus represents the (***outdated***) history of the virtual global module. In Section 8.4.5 and Section 8.4.6 we will see how the *history cache* is filled and used.

Storage Space Consumption

After each global pull (see operation *O-6*) or push (see operation *O-9*) the local and the peer-to-peer module have the same content. PlatinVC does not require twice the amount of storage space for the local and peer-to-peer module, as Mercurial handles the modules using ***lazy copies***. The complete version history of a single artifact is stored in a file[3]. Hard links are used to avoid wasted hard disk space for identical files: whenever a file in a module is an identical copy of a file in another module on the same machine the actual data is only stored once on the hard disk. Hard links point to this data from within the respective modules.

The history files in the local and peer-to-peer module differ only in two cases: When snapshots are created locally and not pushed globally and when snapshots of other peers are received without being requested by the user of the local peer (in which case his machine is the maintaining peer for the received snapshots). In the first case the repositories have the same content after the snapshots are pushed again using operation *O-9: Push globally* (the pull operation does synchronize the modules), in the second case the snapshots are equal when the user pulls those snapshots (which will be very fast, as they are already locally present). After pushing the locally recorded snapshots globally the modules share the same hard disc space again. This behavior is realized by executing the 'hg relink'[4] command, which tells Mercurial that two modules are identical, so they can be stored using hard links like done when a module is cloned.

The operation *O-6: Pull globally* command pulls snapshots from other peers into the peer-to-peer module first and into the local module directly afterwards. If the local peer is responsible for tracking the changes of some folders, as detailed in Section 8.4.2, other peers will push

3 As long as the file is not renamed or moved. In this case a new file is created to store all new versions
4 See http://mercurial.selenic.com/wiki/RelinkExtension

their snapshots to the local peer's peer-to-peer module. These snapshots are not applied to the local module, so the updated history files consume additional space until the local user updates her local module executing operation *O-6: Pull globally*.

In any case the modules can be synchronized by issuing a push and a subsequent pull operation, but the difference might be intended. Normally unpushed changes from the local module should be collected until the latest snapshot does not break the project. Likewise globally pushed changes should not be applied to the local module unnoticed by a user, as this will confuse the local history. If desired, however, changes could be exchanged automatically to mimic a behavior more similar to centralized version control systems with minimal changes to PlatinVC (more about this can be found in Section 11.3).

OVERVIEW OF STORED MODULES For each project a developer has a separate working copy and an accompanying local module. On her machine exist at least the same number of peer-to-peer modules as local modules, which do consume extra space only for differing files, i.e., when additional versions exist for some artifacts in one of the modules. As explained in the following section her machine might by responsible for any folder in any other module, depending on the assignment according to the identifier space. If that is the case the modules containing the other folders are stored on the developer's machine as well, without an accompanying local module and working copy. If the developer's machine is not any more responsible for maintaining a folder in a project she does not work on, the respective peer-to-peer module could be deleted. To avoid the time and bandwidth costs needed to copy a module, if the machine becomes responsible again, it is best to not delete these peer-to-peer modules.

The more modules (or more precisely the more folders) are in the system, and the less peers are online, the more likely all peers store all modules. If the modules can be clustered in sets, so that no developer works on a module which is part of a different set, it is better to have separate networks. This avoids developers having to store modules that they do not work on on their machine.

Analysis of Design Alternatives

All of the related peer-to-peer approaches examined in Chapter 5 store an artifact's history in a single module. This saves storage space, but the resulting drawback is that local snapshots from the user are mixed with global snapshots of other users. The two step commit process that fulfills requirement *R-1.9: Enable local commits* is not possible with a single module. Pastwatch (see Section 5.3.5) separates a user's local module from a global module, but only to be able to offer requirement *R-8: Offline version control*. The local module is updated automatically with the latest changes in the global module. In Pastwatch every peer has a local module, but only some have a global module.

8.4.2 *Repository distribution and partitioning*

The ***global repository*** stored in PlatinVC is distributed ***maintainer based*** among specific peers, following the mapping rule used for ***distributed hash tables***. Snapshots are sent to a limited number of peers, which can be executed in a short time. In contrast, if the repository would have been ***distributed replicated*** among all peers the time needed to share a snapshot would be significantly higher. Nevertheless, PlatinVC shares some properties of a replicated repository. Whenever needed a developer retrieves missing snapshots. Often a developer is interested to retrieve all snapshots existing. If all participants do so the snapshots eventually exist on all peers. In this way updates to a repository are initially shared with only a few peers, so they are retrievable in a short amount of time without being affected from the network's size, but they are distributed eventually to all participants, making them highly available (using ***owner path caching***) and robust to go missing.

All artifacts that belong to a project are stored inside an outermost folder in a working copy. Typically parts of a project, like packages that implement certain functionalities, are organized in folders. We exploit this folder structure to partition a project among the maintaining peers.

The repository is distributed as follows among the peers: In each module an identifier is calculated for each folder by calculating the ***hash value*** of the folders name and path from the uppermost directory in the working copy. The path is a string that consists of a module's name and the names of all folders, which have to be traversed to reach the named folder.

The used ***hash function*** is the same the peer-to-peer overlay uses to compute a peer's identifier. In the utilized Pastry [RD01b] implementation FreePastry [FP] a hash value is calculated using the SHA-1 algorithm [Bur95], which maps any string to a 160bit, represented by a 20 digits hex number. Mapping a folder's name and path to this number allows us to assign the folders to peers, as the domain for both, the folders' and the peers' identifier, is the same. A peer whose identifier is ***numerically close*** to a folder's identifier takes the role of the ***maintainer*** for all artifacts residing in this folder. The numerical closeness is measured in Pastry by comparing the peer's identifier's digits with the folder's identifier's digits. Among the online peers the one whose identifier shares the longest uninterrupted sequence of digits is responsible for the folder with this identifier.

The folders of all modules in the repository follow a normal distribution among the peers, because the hash values are calculated using a user's name (for the peer's identifier) and a folder's name, path and module are normally distributed values in the above mentioned domain.

A **maintainer** is responsible for tracking snapshots of modified artifacts, which reside in an assigned folder. Whenever a peer executes operation *O-6: Pull globally* or operation *O-9: Push globally* a maintainer is contacted. Let us assume that a developer made changes to artifacts in two folders and recorded them in a local snapshot. When she pushes this snapshot globally it is sent to the maintainer of the respective folders. Both maintainers apply the same snapshot to their peer-to-peer module. As detailed in Section 8.4.4 these maintainers store the global latest versions until snapshots are committed that contain modifications on artifacts in folders that are not maintained by those peers.

Which concrete peer is to store which data depends on the identities of the peers currently online in the network, as in any ***DHT*** based storage. The fewer peers are online the more likely a peer is responsible for multiple folders. The latest snapshot a maintainer stores is not necessarily the globally latest snapshot that exists. It is assured that the latest versions of the artifacts in the maintained folder are in the maintainer's peer-to-peer module, but more recent snapshots might exist as well. If a maintainer is responsible for other folders, as a maintainer or a replicating peer as described in the next section, all snapshots of the artifacts in those folders are present as well, which might be more recent. As already described, it is hard to predict which folders a peer will be responsible for and this assignment changes dynamically initiated by joining and leaving peers. Thus the minimum guarantee given is that all versions of the artifacts in the maintained folders are locally present. As detailed further in Section 8.4.4 these additionally stored snapshots are not retrieved. Sending them to a peer which expects updates on artifacts included in a specified folder would only disable the automatic isolation of concurrent work (see Section 7.3.1).

Analysis of Design Alternatives

A number of alternatives have been discussed in Section 6.5.3. For all solutions, in which the identifier of an artifact's history cannot be computed using local information, such as the artifact's name and path, an additional ***index*** list is needed. The major disadvantage of Pastwatch is such an index list, which is stored by one peer and accessed all the time by all other peers. This list contains the names of the members of a project. All snapshots committed by a member can be found on the peer responsible for the hash value computed using the members name.

Peer-to-peer based version control systems, which partition the repository similar to PlatinVC, assign the history of single artifacts to specific peers. We decided to distribute the

repository less fine grained, so that the number of maintaining peers is smaller. This results in a smaller number of peers involved in the global push operation, which is shown in the fewer transferred messages.

Assigning an entire module to a single peer is too coarse grained. The push and pull operations would be more efficient, as only one peer has to be contacted, but this one peer would be contacted by every developer working on the maintained module. Many of the drawbacks of having a centralized solution (see Section 2.4.1) would reappear.

8.4.3 *Replication*

To counter failing peers all relevant information in the system exists on multiple peers. Following the ***DHT*** approach a subset of a peer's ***neighborhood peers***, called the **replica set**, replicate all relevant data. The replication has to be as reliable as possible. Failing peers are immediately replaced, as elaborated in Section 8.5.

A ***maintainer*** is concerned with replicating updated data to its replica set. When new data is about to be stored by the maintainer, which is only the case during a global push, it sends the data directly to all replicating peers. Only if all replica peers acknowledge that they stored the data it is treated as applied and the user is informed. Different situations, which can occur depending which peer is failing, are analyzed in Section 8.5.

The data that needs to be replicated is reduced to a minimum in order to avoid the costly (in transferred messages and needed time) update process described in Section 8.5. The maintained partial repository, consisting of the modules that include the folders a peer is responsible for, is replicated. A peer can be the maintainer for some folders and a replica peer for other folders at the same time. Regardless of why a folder is stored with the latest versions of the included artifacts, they are all stored in the same ***peer-to-peer module***. When a developer asks for the latest snapshot regarding the artifacts in a specified folder, however, no newer snapshot is sent which includes changes of artifacts outside the specified folder exclusively.

The repository's ***history cache***, needed by the collaboration procedure detailed in Section 8.4.4, is replicated as well. It is updated whenever a maintained module is updated. Thus it is sent as additional data during the replication of a module's update and does not require additional messages.

All status information is retrieved by a message type and tracked by a service taking peer only. For example, if a peer requests to push changes, a maintainer sends an allowance. The maintaining peer does not remember that it has sent an allowance, nor does it count for a timeout to resend it. If the answer is lost, the requesting peer will resend its request. When the requesting peer proceeds with sending new snapshots, the maintainer knows from the message type how the message has to be handled. All messages have a unique session identifier, which is remembered by the requesting peer only. A maintainer sends the session identifier that it took from the message it is answering, it does not remember it from the first received message in the push protocol. If a request is resent and the requesting peer receives two answers (e.g. the first answer was received delayed, after it resent its request), the requesting peer can recognize the duplicated message from its session identifier.

As already mentioned, only the requesting peer remembers the phase of a push or pull protocol. The contacted peers only react to the received messages. Only the requesting peer waits for timeouts and resends its request in case a message is lost or the receiver failed. If the requesting peer fails it is not necessary (or possible) to complete the protocol, thus the status of a request does not have to be replicated.

Analysis of Design Alternatives

In some systems like Scalaris [SSR08, MH07] the peer who pushes snapshots updates the replicas as well. While the routing takes care to find the maintainer of a folder specifying the folder's identifier, the identifiers of the replicating peers have to be requested first. By the

time an answer is retrieved they could be replaced already. Only the maintainer is updated with changes among the replica peers through the ***maintenance mechanisms***. This alternative would be more error prone and would not speed up the push operation, as the identities of the replicating peers would have to be requested first.

8.4.4 *Collaboration*

The *share* and *retrieve* operations listed in Section 7.4 diverge from each other in the grade of decentralization. When issuing the update operation a user wants to obtain ***global knowledge***, i.e. the latest snapshot among all participants.

When a user shares a snapshot he is not very interested whether other users receive it. Likewise a user is not interested in all shared snapshots but usually only the latest. It is often only important to know if another user modified the same artifacts, which could lead to a conflict. Therefore the push operation needs this minimal global knowledge, which is needed by the respective authors only, rather than globally by all participants.

Global Knowledge

The biggest difference between centralized and decentralized solutions is the degree of global knowledge they have. Because all information is gathered in a central place, a centralized solution has complete global knowledge. The knowledge in a decentralized solution has to be aggregated with specific mechanisms and is partitioned among the participants.

An example for a system, which has *complete* global knowledge is a mainframe system [WB96], where users operate through terminal programs, which run on a central instance.

In a client-server system a user has to first submit his local information so that it becomes part of the global knowledge collected on the central server. Thus the global knowledge is not as complete as in a mainframe system.

There are mechanisms for peer-to-peer systems, which gather and update the distributed knowledge, like the update mechanism presented in this work. They are always designed for a specific application.

One example of a system with much less global knowledge is any distributed version control system presented in Section 5.2. Here a user has to actively exchange his information with other users on a one-to-one basis. Eventually the knowledge spreads to all participants, but might be ***outdated*** before doing so.

This consideration resulted in two different mechanisms for sharing and retrieving snapshots. During the several developed iterations of PlatinVC it turned out that optimizing one mechanism immediately results in the worsening of the performance of the other mechanism. The presented solution represents the best trade-off, favoring the retrieval mechanisms as they are typically executed more frequently.

The following sections describe the operations, which are executed in a distributed fashion among several peers. They are explained using UML 2.2 sequence and activity diagrams [UML09]. Prior to the elaboration of the retrieve and share operations an example is given.

Send Message Reliable

Before we elaborate the retrieve and sharing protocols we first take a look at how messages are sent reliably using the unreliable peer-to-peer communication PlatinVC is based on. The presented protocols implement design principles *D-3: Do not rely on any message transfer* and *D-2: Do not rely on the availability of any machine*.

A message can be sent in two ways: If the physical network address of a peer is known, the message can be sent directly to that peer. Otherwise it has to be ***routed*** in the overlay network by a group of peers. We first observe this mechanism on a directly sent message.

SEND A MESSAGE DIRECTLY In any network messages can get lost. The loss rate depends on the physical network the peer-to-peer overlay network operates on. PlatinVC can be deployed on different networks, e.g., wide area networks (WAN) like the Internet or wireless local area networks (WLAN).

Several situations can be observed during a message transfer, depicted by the alternative sequences in the sequence diagram in Figure 30. Additionally, this sequence diagram shows the basic collaboration mechanism in PlatinVC: A requester triggers the execution of a mechanism on the receiver's side and receives a computed result as a reply. The only case in which the

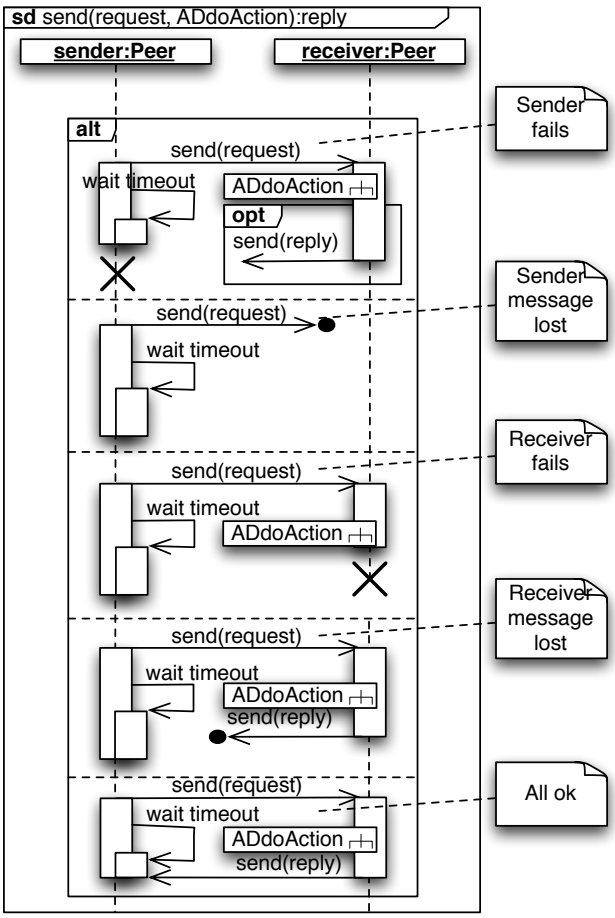


Figure 30: Sequence diagram showing basic message exchange using the physical transport layer

send process is successful is displayed in the last slot of the *alternative fragment*, which is annotated with *All ok*. The sender sends his initial request and starts to count for a timeout. The receiver reacts upon receiving the message by executing some action, which is referred to by the activity *ADdoAction*. The result of this activity is returned to the sender in form of a reply message. If this message is received before the time elapses, the exchange was successful. In all other cases some messages are not received.

The sending peer can fail before it can receive a reply, the sent message can be lost and never reach its destination, the receiving peer can fail before it can send a reply or the reply message can be lost. In the first case the sender is expected to send her initial message again, once her machine is ready. Depending on the concrete action the reply will be computed again or just resent. All other cases are not distinguishable from the perspective of the sending peer. In all those cases the faulty behavior is surmised after a timeout passed (everything might even be ok, if the timeout value is set to low).

If messages were lost, resending the initial request could bring success. If the receiving machine failed, on the other hand, no resent messages will be received. Therefore the number

of retries should be limited. The sequence diagram in Figure 31 presents our implementation. If no reply was received the initial message is resent `retries` times, which is a configurable parameter, set to 3 in our prototype. If no answer is received after the last retry an exception is

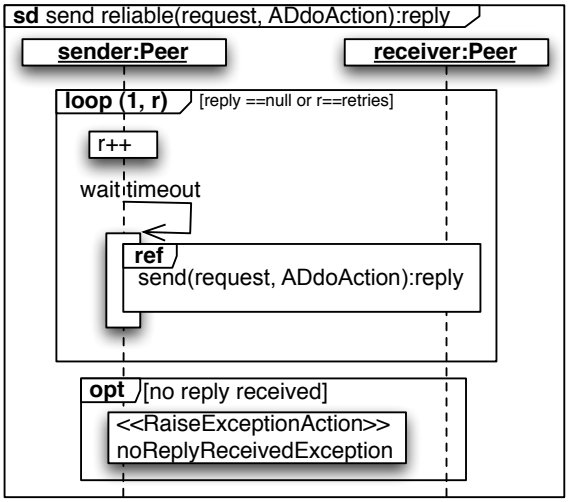


Figure 31: Messages are resent if no reply is received as shown in this sequence diagram

thrown. This ultimately informs the sender to retry the operation at a later time, when the receiver might have recovered.

ROUTE A MESSAGE RELIABLE If the sender did not contact the receiver for a long time it might not have the receiver's physical network address. A message needs to be routed to the receiver first, using the ***routing mechanism*** of the peer-to-peer overlay. Figure 32 describes how we are using this mechanism. A request is sent with a ***peer-to-peer identifier***

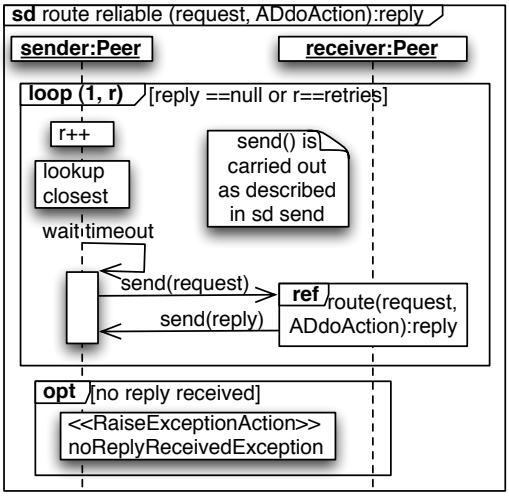


Figure 32: If the direct address of a peer is unknown messages are routed as described by the depicted sequence diagram

as a destination. This identifier can either identify a peer or a resource offered by that peer. The ***DHT mechanism*** takes care to deliver a message which is sent by specifying a resource identifier as the destination to the resource offering peer. The sender sends an initial message to the peer who is ***closest*** to the destination peer-to-peer identifier, among the ***neighboring peers***. When a peer is a **neighbor** its physical address is stored in a routing table and mapped

to its peer-to-peer identifier. The routing table entries are kept up-to-date according to the update policy of the ***peer-to-peer protocol*** with periodic ***maintenance messages***. When a peer receives the message it looks in its routing tables for a peer which has an identifier closer to the destination. If it turns out that no other peer is closer it accepts the message, executes the specified activity (as presented by *ADdoAction*) and sends the resulting reply. If there is a closer peer the initial request is forwarded to this peer recursively. The returned answer is sent directly to the original sender, who added her physical address to the request.

The destination of the request is a peer-to-peer identifier, which will be received by the peer, whose identifier is closest among on online peers. If the receiver fails the ***routing mechanism*** will automatically deliver the message to a substituting peer. Thus, even if the receiver fails, the message exchange will be successful. Nevertheless the message itself can fail at any of the multiple intermediate transfers. Thus the possibility of a failing message is multiple times higher than in the previously described direct message exchange.

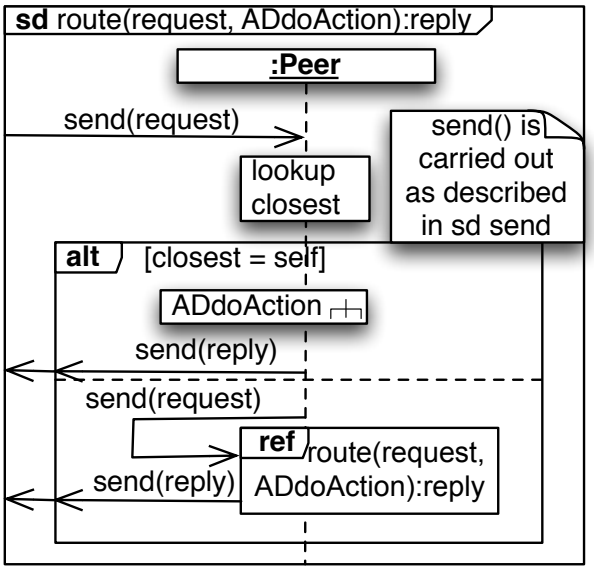


Figure 33: Similar to the resending presented in Figure 31 routed messages are repeated as well

To ensure that a message eventually reaches its final destination it is resent in the case a reply is not received within a predefined timespan, similar to the protocol presented in Figure 31. Figure 33 presents its adaptation to the route protocol. Only the original sender repeats the operation if the timeout is passed, as it would introduce to many messages, if the intermediate peers would resend messages as well. It is sufficient if the initial sender ensures messages are resent. If the timeout elapses before the original sender receives a reply she resends the message, which triggers resending the message by the intermediate peers. The number of `retries` can be very high, in contrast to the retries of the direct message exchanges described before. The destination of the request is the closest peer to the identifier set as the message's destination. Thus, if the actual peer in this position fails after the message was sent, its closest neighbor will receive and handle the message (see ***routing mechanism***).

The timespan to wait for a reply has to be greater than in the case of a direct sent message, as the timespan a message needs to arrive at its destination depends on the overlay network structure, which changes dynamic when peers are joining and leaving. A message routed in the overlay network is even less reliable as it consists of multiple received and sent messages. If any of those are lost the original message will not reach its final destination. The number of intermediate peers, called **hops**, depends on the momentary structure of the peer-to-peer network as well.

ENSURE PROGRESS DESPITE FAILED COMMUNICATION As we saw before a message transfer can eventually fail. A routed message could be repeated unlimited times, and will eventually be received by a peer able to compute a reply. But a directly sent message cannot be delivered if the receiver fails. The initiating operation has to be aborted when a directly sent message could not be delivered at the last retry. Following design principle *D-5: Avoid storing statuses* only the initiating peer takes care of the progress of an operation. This is always the peer of a developer, who wants to execute any of the operations PlatinVC offers. The executed activity *ADdoAction* is always initiated by the received request. The executing peer does not track a state to know which message is expected next, but reacts on incoming requests only.

In this way only the sender actively triggers the phases in an operation. If a message transfer fails, as depicted in the alternative fragment in Figure 30, the receiver's activities would not be triggered. This behavior has been considered in the design of PlatinVC, as we will see later. No activity leads to faulty states when it is nor continued by a follow-up activity. In this first case the sender would repeat the operation after it rejoins the network. If the operation was actually finished, and just the final acknowledgement was missing, only this acknowledgement is sent. As an example we take a look at the push operation detailed further in Section 8.4.6. The push operation is, like the commit operation, *idempotent*, meaning that executing it a second time does not alter the repository. If the local committed snapshots have been already transferred and applied, the maintaining peers will notice that no snapshot needs to be transferred again. If they have not been applied yet the complete operation has to be repeated. Applying a snapshot is an *atomic* action. However, it can happen that not all maintainers can apply a snapshot. Depending on the actually contacted maintainer a pull might not retrieve the latest snapshots. In this case still ***causal consistency*** is guaranteed. A repeated push of the original sender, or a push of any peer who updated the latest snapshot before from a correct maintainer, will repair the incomplete repository state.

8.4.5 *Retrieve Updates*

Before we elaborate the operations in detail an example should illustrate the following protocol.

Retrieve Update Example

Let us assume that the project, and thus an updated working copy, has the structure shown in Figure 34. In the outermost folder, labelled with '/', two folders are contained. In the first folder 'docu' are the artifacts 'Plan.csv', 'Readme.txt' and 'Manual.txt'. All those artifacts belong to the documentation subproject. The other folder 'impl' contains an artifact named 'Main.java' and a folder 'utils', which contains the artifacts 'Helper.java' and 'Parser.java'. All folders are named with small letters, artifact names start with capital letters.

Alice is about to *pull* the latest versions from the subproject 'impl'. She is not interested in updates to the artifacts in the 'docu' folder. Let us assume that her local and peer-to-peer module store identical snapshots as presented in Figure 35. The recorded snapshots are presented as boxes, pointing to their parent snapshot. They are labelled with their identifiers (shortened to three letters), which are the ***hash values*** of their content. Thus they are in no numerological order (it would be hard to maintain a globally continuing value among all peers). In this history two branches are visible, ending with the snapshots C87 and D34.

Alice executes the operation *O-5: Pull globally using specified folders*, specifying that only updates from 'impl' are desired. In the first step Alice's peer searches for the peer responsible for maintaining updates on this folder. It sends the identifiers of known snapshots so that the contacted peer knows which snapshots are missing. The latest snapshots are C87 and D34. These snapshots do not contain versions of artifacts which are requested. The maintainer for the desired folder 'impl' might know about these snapshots - but this cannot be guaranteed. It is only certain that this peer knows all snapshots containing versions of artifacts residing in the folder 'impl' or any of its subfolders, being the folder 'utils' in this example only. The parent of the snapshot D34 is B56 and contains a new version of an artifact located in the

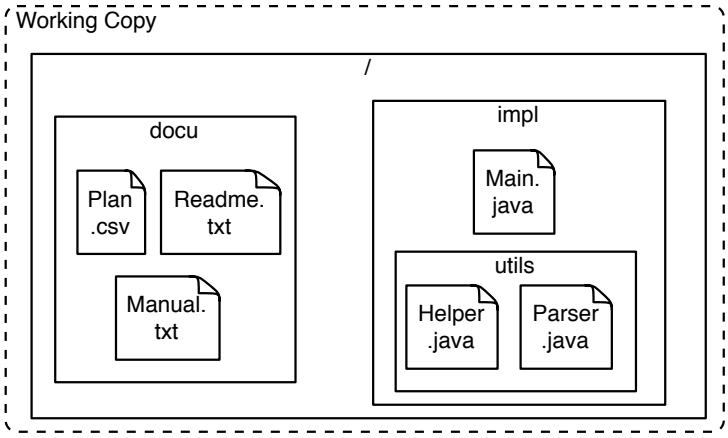


Figure 34: Structure of the exemplary project

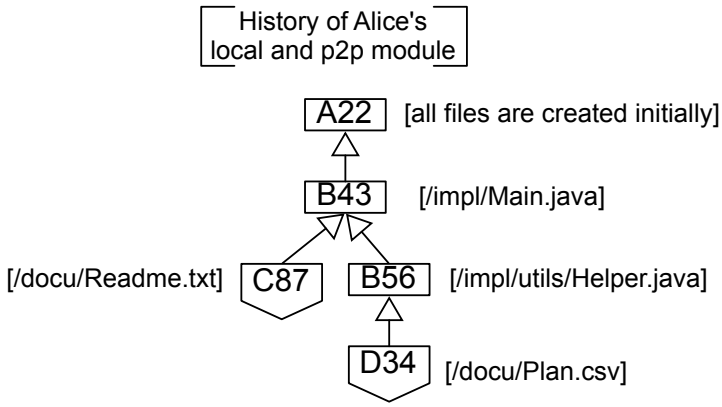


Figure 35: The history in Alice's local and peer-to-peer module

folder 'utils'. The parent of snapshot C87 is B43. It contains a new version of an artifact in the specified folder hierarchy as well, but as it is a predecessor of the snapshot B56 there is no need to add it to the snapshots that should be transferred. The maintainer surely has this snapshot. To disclose which snapshots are stored locally only the identifier of snapshot B56 would have to be transferred. The identifiers of the locally latest snapshots C87 and D34 are transferred as well, as doing so does not bring too much overhead and spreads the information of the snapshots existence.

Let us further assume that Bob's peer is responsible for maintaining the versions of all artifacts residing in the folder 'impl'. The ***history*** of Bob's local and peer-to-peer module are presented in Figure 36. For simplicity we again assume that they store the same snapshots. The snapshots A22, B43 and B56 are identical to the respective snapshots in the history of the corresponding module on Alice's peer. Bob receives the Snapshots C87, D34 and B56. If only the latest snapshots from the branches in Alice's history were sent, Bob would not know which other Snapshots Alice has. The snapshots C87 and D34 are unknown to Bob, so they are applied to the ***history cache*** of Bob's peer-to-peer module. This history cache stores the structural information (i.e. ***metadata***) of the snapshots only, containing the snapshots' identifier, its parents, its author, modified files and the creation date, but not the actual modifications. Bob's peer can compute that Alice does not have the snapshots F78 and 56C, because their identifiers were not sent by Alice. It further computes that all snapshots between B56 and 83A are missing, being snapshot 56C in this example. The parent snapshot of snapshot F78 is known to Alice as she disclosed that she has the later snapshot B56, which was built

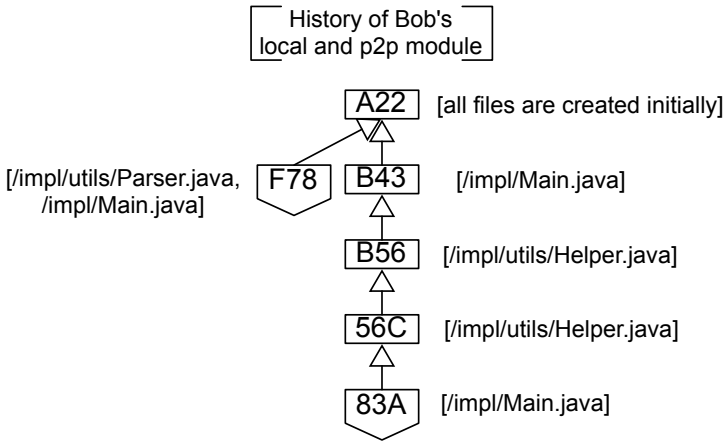
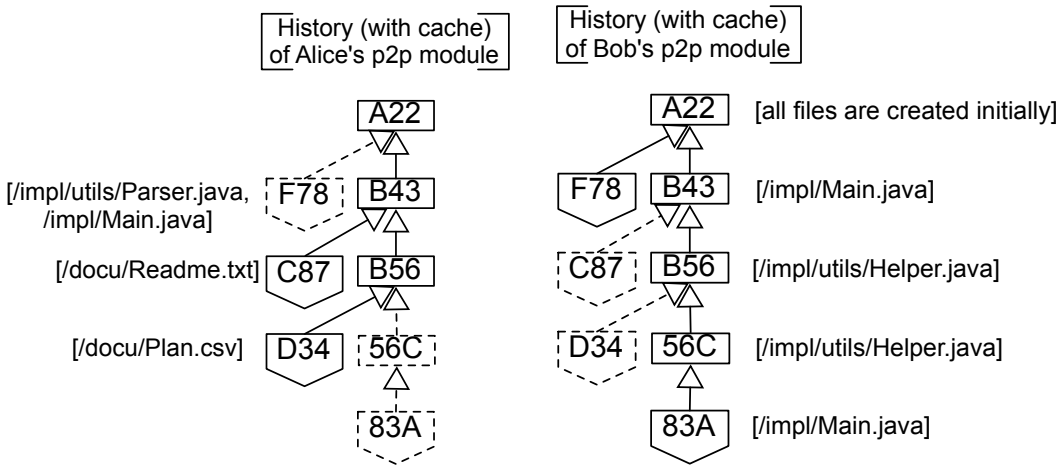


Figure 36: The history in Bob's local and peer-to-peer module after executing the pull operation



(a) on Alice's peer-to-peer module

(b) on Bob's peer-to-peer module

Figure 37: History (cache marked with dotted lines) after the first pull phase

on the snapshot A33. The ***history cache*** entries sent by Bob include the snapshots missing in Alice's history cache, being the snapshots 56C, 83A and F78.

After this exchange the local modules on both peers are still unchanged, but the history cache of the peer-to-peer module stored on Alice's machine contains the new entries, presented in Figure 37a. The history cache entries, which do not exist as snapshot in the peer-to-peer module, are marked with dotted lines. Bob's peer-to-peer module is shown in Figure 37, the entries which exist as metadata only are marked with dotted lines as well. These are the cache entries which were sent by Alice.

In the second phase of the two step pull operation Alice now chooses a minimal number of peers needed to be contacted in order to retrieve the latest versions of the artifacts in the folder 'impl' and its subfolder 'util'. The snapshots Alice is interested in are the snapshots F78, 56C and 83A. Alternatively Alice could choose herself which snapshots she would like to pull, but considering a workflow, where developers like to work on separate branches, as presented in Section 7.3.1, the aforementioned snapshots are automatically chosen. Snapshot 83C is based on snapshot 56C, thus the latter is automatically pulled when snapshot 83A is. In snapshot 83A an artifact in folder 'impl' was modified, snapshot F78 contains a new version of an artifact in folder 'utils' and a new version of the artifact in folder 'impl'. Therefore all

snapshots can be pulled from the maintainer of the folder 'impl'. If in snapshot F78 the artifact Main.java had not been modified, and thus only an artifact in the folder 'impl/utils' were changed, Alice would have to request the snapshot F78 from a second maintainer. In the worst case a maintainer has to be contacted for each branch, regardless of the number of updated folder. In the best case, when only a single artifact was modified in the latest snapshots of all branches, one maintainer can provide all missing snapshots.

The actual request message contains all desired snapshots, being F78, 56C and 83A in our example. It is routed using the peer-to-peer overlay network to the responsible maintainer. Assuming that 'Main.java' has not been changed in the snapshot F78 two identical messages would be routed to the maintainers of folder 'impl' and 'utils'. The snapshot B56 which is present on both maintainers would only be send by the maintainer of 'utils' to avoid unnecessary data transfer. If one peer were responsible for both folders the second message is discarded (not answered, nor repeated).

Following the ***peer-to-peer routing mechanism*** a receiver forwards the request to the next closest peer. Prior to forwarding a message the receiving peer checks if it can answer the request itself, at least partially. If it can it sends the requesting peer, Alice's peer in our example, some of the requested snapshots and modifies the forwarded request to only contain those not already sent.

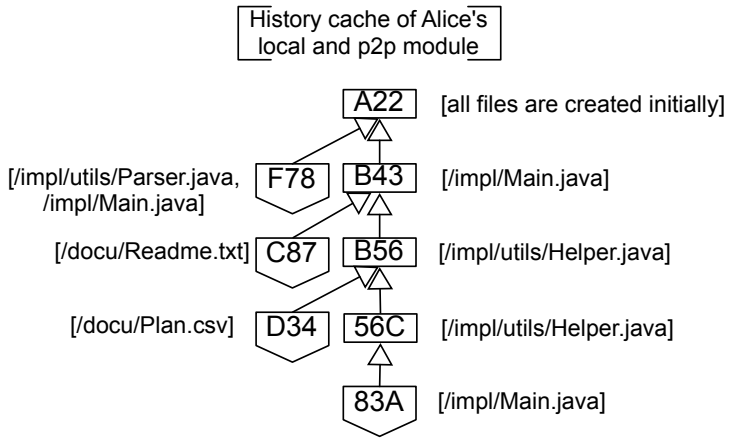


Figure 38: The history in Alice's local and peer-to-peer module after executing the pull operation

After Alice applied the requested snapshots her local and peer-to-peer module appear as shown in Figure 38. Bob's local module remains untouched as presented in Figure 36, his peer-to-peer module does not receive new snapshots, but the history cache is updated, as illustrated by Figure 37.

Retrieve Protocol

The pull operation retrieves the latest updates, which consist of all locally missing commits pushed by coworkers. These commits have been applied to the peer-to-peer module of a number of maintainer by the share operations (see Section 8.4.6) and are pulled from there to obtain the latest existing snapshots.

There are three variants described in Section 7.4.2. Operation *O-7: Get Artifact* and operation *O-6: Pull globally* are variants of the operation *O-5: Pull globally using specified folders*. Operation *O-7* is implemented by issuing operation *O-5* specifying the folder in which the requested artifact resides. Operation *O-6* is implemented by executing operation *O-5* specifying the outermost folder (or all folders, which has the same effect).

The approach presented in this work has been optimized in favor of the pull operations. In a usual development workflow artifacts are more frequently updated than changes are committed. The result is a two step process, in which first structural information about existing

snapshots, called a repository's ***history cache*** **entries** in the following, are retrieved. In the second step a maintainer is contacted, who has all needed snapshots. For each specified folder all artifacts in this folder and its inner folders are updated. When folders are specified which are subfolders of one of the specified folders the operation is executed only for the outermost folder. The maintainer of the outermost folder does not necessarily store all snapshots of the artifacts in the subfolders - but it stores the ***history cache entries*** of this snapshots, so that their existence is known and they can be retrieved from the subfolder's maintainer.

This behavior, however, could be changed easily. We chose to implement this behavior to exploit the natural partitioning of a project into subfolders. We assume that in most cases a developer is interested in the latest version of all artifacts beginning from a specified folder. In contrast to our solution in most existing solutions updates on all existing artifacts are always retrieved, which is equal to pulling the latest versions specifying the outermost folder of a project in our solution. This enables a project structure, where multiple different projects reside in a working copy, stored under a common outermost folder. Snapshots would be created considering all existing artifacts, but updates could be pulled for single projects only. N.B. ***Subversion*** offers a similar functionality when subfolders are specified as a repository's root directory.

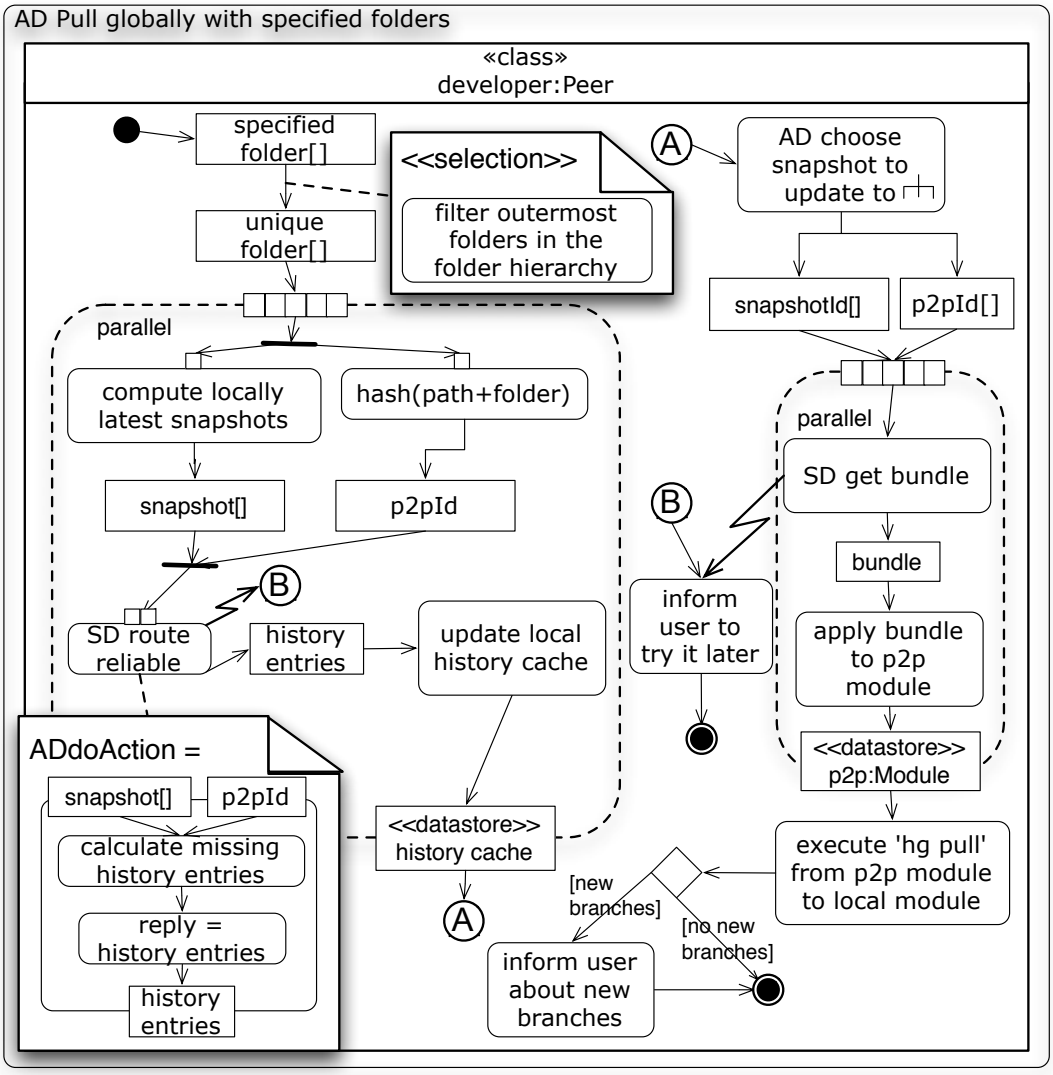


Figure 39: Activity diagram of the pull operation

The activity diagram in Figure 39 describes the pull procedure of the operation *O-5: Pull globally using specified folders*. A user's *specified folders* are first *filtered*: Each folder which is the child of another specified folder or its children in the directory tree is removed. Updates in these folders can be retrieved by communicating only with the maintainers responsible for the outermost folders among the specified folders. The filtering process results in folders, which are in parallel branches of the directory tree, called *unique folders* in Figure 39. The next procedure runs parallel to each specified unique folder.

This parallel activity, described in the left side of Figure 39, is the first step. In this step a module's ***history cache*** is updated to reflect the latest versions with respect to the specified folders. The locally latest snapshots in all ***branches*** (named and unnamed ones) from the peer-to-peer module containing the specified folders, are retrieved (in the action *compute locally latest snapshots*). If the folders belong to more than one module the whole operation is executed for each module independently. The resulting set contains the most recent snapshots in all branches up to the parents of the latest snapshots, where artifacts residing in the specified folders or any of its subfolders were changed. As detailed in the push operation description in Section 8.4.6 we can expect the maintainer of the specified folder to have those snapshots, but not necessarily later ones. These snapshots are needed to determine the minimum number of snapshots that need to be transferred to a developer in order to update his peer.

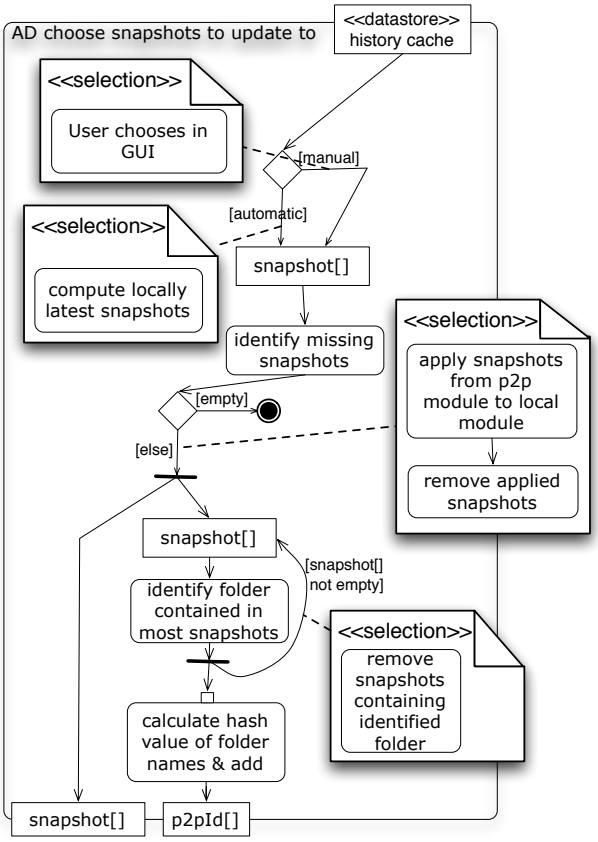


Figure 40: Activity diagram: Choose the snapshots to which the local repository should be updated to

At the same time the identifier of the maintaining peer is computed by ***hashing*** the module's name concatenated to the folder's path and name. The resulting hash value is the key under which all snapshots of artifacts are maintained, which are stored in the aforementioned folder (as explained in Section 8.4.2). A message is routed to this key following the procedure described in Section 8.4.4 (and presented in Figure 32). In the unlikely case, where not even a

substituting peer can reply with the requested update after a reasonable[5] number of retries the pull operation is aborted and the user is *informed to try it again later*. Upon receiving a message the maintaining peer *calculates the missing history entries* and sends them back to the developers peer, which updates the local history cache.

The transferred entries are the ***metadata*** of the missing snapshots only, which do not contain the actual changes of the modified artifacts. In this process the same history entries have to be identified in the local and remote history cache in order to transfer only the missing entries. Section 8.4.6 details how we achieved this with a minimum number of messages.

Our prototype can be configured to present the updated history to the user who can now explicitly choose snapshots he wants to update. All snapshots starting from the latest local snapshot to the specified snapshot are retrieved. In most cases, however, a user wants to have the latest snapshot including new versions of all artifacts in a folder's hierarchy. Alternative to a human's interaction the snapshot fulfilling this criteria is computed in the activity labelled with *AD choose snapshot to update to* and illustrated in Figure 40.

Similar to the algorithm described in the above example, the latest snapshots in all branches are investigated. If the snapshot does not contain a new version of any artifact residing in any of the folders starting from the specified folder, it is discarded and its parent is investigated, until a snapshot fulfilling this criteria is found for all branches. To find the minimum number of maintainers that need to be contacted these snapshots are examined further. The *identified missing snapshots* are retrieved from the peer-to-peer module, if they are present. Only the remaining snapshots, which are not stored on the local peer, are retrieved in the following steps.

In the following action (*identify folder contained in most snapshots*) the folder which is contained in the most snapshots is searched, regardless which of its artifacts was modified in the respective snapshots. If multiple folders are contained in the same number of snapshots one of them is randomly chosen. The hash value of the folder's name is calculated and added to the list of p2pIds, which is the list of peers that need to be contacted to retrieve all missing snapshots containing the folder. All snapshots containing the identified folder are removed and, as long as the set of snapshots is not empty, the next folder is searched.

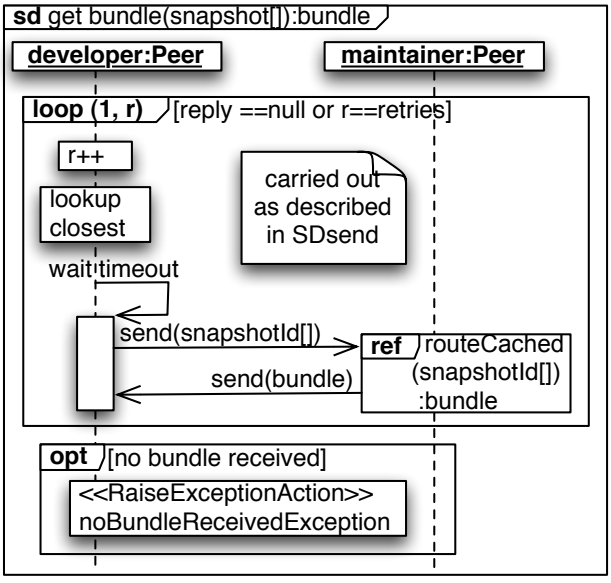


Figure 41: Activity diagram: Get bundle with missing snapshots in the second phase of the pull operation

5 If a receiver fails the message is ***routed automatically*** to a substituting peer, thus the number of retries can be high, as eventually a peer will process the request.

For each maintainer resulting from the filtering process just described a bundle is requested with the activity *SD get bundle* (see Figure 41), which contains the missing snapshots only. The process, which routes a request containing the latest local snapshots (resulting from the previous calculations) to the maintainers is very similar to the activity *route reliable*, presented in Figure 33. The only difference is that any peer on the path to the maintainer can answer the request, even partially, if it stores any of the requested snapshots. The request is modified by removing the snapshots, which were already sent to the requesting peer. Doing so implements **owner path caching** and reduces the load on the maintaining peer the request has been addressed to. A second positive effect is that a request is answered faster. Pastry optimizes its routing tables with the peers to which the message delay is minimal. If those peers answer a request rather than forwarding it to the concrete maintainer the new snapshots are received faster. It would be possible to proactively store frequently requested snapshots. However, we did not implement proactive caching, as the popularity of the snapshots most probably changes too dynamically: Only the latest snapshots would be popular. The user of a peer who is forwarding multiple requests might request those snapshots as well, which effectively caches them. N.B. ***Owner path caching*** is not possible in the other activities, such as *AD get history update*, as this information is only up to date on the maintainer and its replicas and cannot be answered from a copy.

In a last step a retrieved bundle is *applied to the peer-to-peer module*, from where it is *pulled to the local module* using Mercurial's 'hg pull' command. A *user is informed about new branches*, if they resulted from applying the retrieved snapshots, so that he can react and merge them. If she decides to leave the branches unmerged future updates are applied to those branches without notification.

ON CONSISTENCY For each specified folder either the latest snapshots are received or none. Thus it can happen that updates for some folders are received, but not all, resulting in ***causal consistency***. If a message is lost it is sent again `retries` times. If the procedure can still not finish successfully, which can happen when a lot of participants are leaving and joining the network (i.e. high churn), the pull operation is aborted and the user should execute it at a later time. In most cases, investigated further in Chapter 10 as well, updates for all folders are received, resulting in ***sequential consistency***. Regardless of the network conditions, if only updates on folders, which are included in the hierarchy of one of those folders, are pulled and in the latest snapshot of all branches the same artifact has been changed, a single peer can provide all missing snapshots. This is the case when all updates of all artifacts are requested and only one branch exists. In this situation ***sequential consistency*** is guaranteed.

Analysis of Design Alternatives

In a preliminary version of PlatinVC the updates had to be pulled by specifying the folders in which updates should be looked for. This worked well if a user was interested in updates of a set of folders, working on automatic isolated branches as described in Section 7.3.1. The push operation was a bit more performant then the implemented variant presented in this work, at the cost of a less performant pull operation. Its complexity rose with the number of folders one was interested to receive an update from. For each folder a maintainer had to be contacted. Although these maintainers were contacted in parallel and depending on the distribution of the peers and the repository a peer could be responsible for multiple folders, lowering the total number of peers needed to be contacted. However, there was another problem as well: A newly created folder could only be retrieved, if the requesting developer knew its name or if in the same or in a child snapshot an artifact in a known folder was changed.

Not only to solve the problem of unknown folders but to improve the performance of the pull operations as well, we developed the approach presented in this section.

8.4.6 *Share Versions*

Similar to the presentation of the retrieve operations the share operations are introduced with an example, before being detailed using activity diagrams.

Share Versions Example

We take the same project structure presented in Figure 34 in this example. Let us further assume that the resulting local history cache of the peer-to-peer module on Alice's and Bob's peer is the result of the previous example, presented in Figure 38 regarding Alice and Figure 37 regarding Bob. The snapshots stored in Bob's local module are illustrated in Figure 42. It

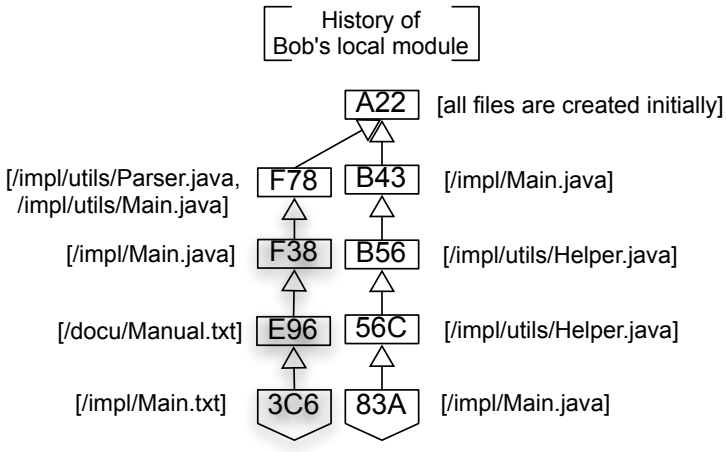


Figure 42: Exemplary history in Bob's peer-to-peer module

does not contain the updated history cache entries C87 and D34, which were received from Alice and added to the history cache of Bob's peer-to-peer module (compare to Figure 37. Bob recently committed the snapshots F38, E96 and 3C6 locally and is about to shares them now executing the operation *O-9: Push globally*. He first changed the file '/impl/Main.java' in snapshot F38, then '/docu/Manual.txt' in a subsequent snapshot E96 and 'impl/Main.java' again in the last snapshot 3C6. In order to share this snapshots three other peers have to be contacted. As we have already seen specific peers are responsible for keeping track of all snapshots that include changes to artifacts in specific folders. In the *push* process Bob's peer first finds out which peers it needs to send the new snapshots. In snapshot F38 and 3C6 an artifact in the folder 'impl' was changed. In snapshot E96 Bob modified an artifact in the folder 'docu'. The parent folder of both folders is the outermost folder of the project '/'. So the maintainers of the folder 'impl', 'docu' and 'root' have to be contacted. As introduced in the previous example Bob is the maintainer for the folder 'impl'. The assignment is computed by hashing the modules name and folder path. The peer ***closest*** to the resulting value is the maintainer for that folder. In our example Alice is responsible for keeping track of the artifacts in the folder 'docu'. An additional peer is responsible for the folder '/', we name it Cliff's machine.

Artifacts in two folders were changed, so Bob's peer ***routes*** two requests to the maintainers of the folders 'impl' and 'docu'. The request for the folder 'impl' is redirected to itself without introducing network traffic. All snapshots are applied to the peer-to-peer module and an acknowledgment is returned, which is received by Bob's peer before being sent over the network.

The other request is directed at Alice's peer. Alice opens a TCP/IP socket for the data transfer and sends its locally latest snapshots, elaborated further in Section 8.4.6, so that Bob's peer knows which snapshots have to be transferred. Again, guarantees about the entries in

Alice's peer-to-peer module can now be given. From the answer Bob's peer finds out that in addition to the recently created snapshots, snapshot F78 is missing as well (only its metadata is present). Only in snapshot E96 an artifact in the folder 'docu' is modified, so only the missing snapshots up to this snapshot have to be transferred. Although snapshot F38 does not contain changed artifacts in the folder 'docu' it has to be transferred to Alice - otherwise ***causal consistency*** would not be guaranteed. Snapshot 3C6 does not have to be transferred. In order to achieve more redundancy and thus a higher robustness snapshot 3C6 is sent to Alice as well. In the unlikely case that all peers responsible for the folder 'impl' fail in a time frame that is too short to replicate the stored snapshots to replacing peers, a later push from Alice based on that snapshot would recover it. Upon receiving the snapshots they are applied

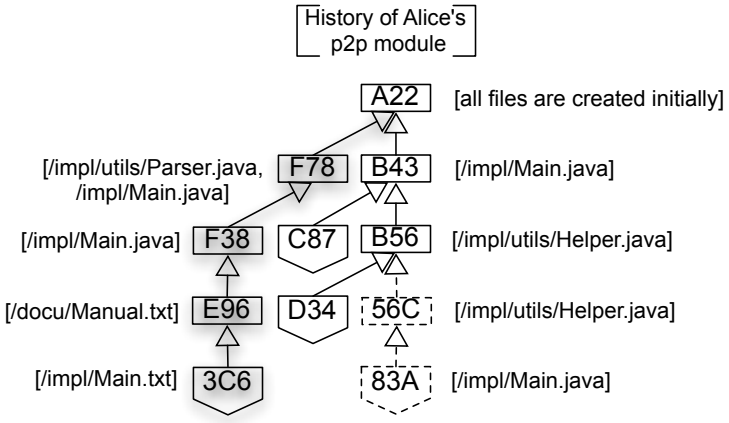


Figure 43: Exemplary history in Alice's peer-to-peer module

to Alice's peer-to-peer module as illustrated by Figure 43. Additional to its metadata form the actual contents of snapshot F78 are stored in Alice's peer-to-peer module now as well.

After applying the received snapshots, but before replying with an acknowledgement, all replicating peers are updated by Bob's and Alice's peer. To allow a faster pull operation, as described in Section 8.4.5, the ***history cache*** of all maintainer responsible for parent folders of the folders with changed artifacts have to be updated. This occurs parallel to the replication. Bob's and Alice's peer compute that the parent folder '//' has to be updated. Both know that both are about to send the updates to the maintainer of the folder '//', being Cliff's peer. To avoid duplicate messages a simple heuristic decides that the maintainer of the folder with the smaller identifier updates the common parent folder only. The hash value of 'impl'[6] is smaller than that of 'docu'[7], so only Bob's peer updates the ***history cache*** of the peer-to-peer module stored on Cliff's peer, which results in the history presented by Figure 44. Only the first snapshot is completely stored, all other snapshots exist as metadata entries only. If parent history cache entries on Bob's peer are missing it requests a full update, as described in Section 8.4.6.

After all subprocesses finish successfully each maintainer sends an acknowledgement to the initiator, Bob. When all acknowledgements are received Bob's peer-to-peer module is updated with the published snapshots as well. If in spite of all retries any subprocess fails to complete the whole operation is repeated by the user. When snapshots are to be resent to a maintainer, who successfully applied them the last time, it simply replies with an acknowledgement, indicating that everything is ok. This shortens the duration of the new attempt and reduces the involved bandwidth consumption. N.B. The local module of the maintainers, Alice's and Cliff's peer, remain untouched by the push operation. If their user wants to get the committed snapshots they execute the pull operation described in Section 8.4.5. This operation would still ask for the most recent updates in the first phase, but the already locally present snapshots

6 SHA-1_hash("impl") =846cfd95cebc380bff149da6c040e3517ea46c4e
7 SHA-1_hash("docu")= 9d86a6f5fa69acfda6ae366bea9a91795ff78eb6

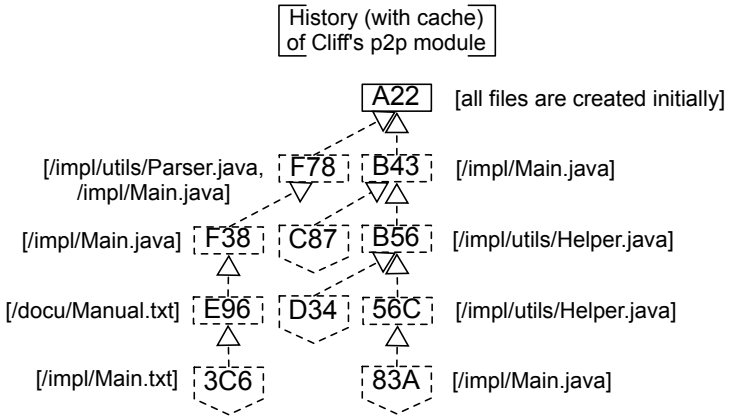


Figure 44: History (with cache marked by dotted lines) of Cliff's peer-to-peer module

would be retrieved without further messages from the peer-to-peer module (initiated by the *identify missing snapshots* activity in Figure 40).

Share Versions Protocol

When a user want's to create a snapshot to record her latest changes in the working directory she either calls Mercurial's 'hg commit' commando or operation *O-8: Commit*, which executes the aforementioned command. This creates a snapshot which is stored in the local module. When the user is ready to share all her locally stored snapshots she invokes the operation *O-9: Push globally* of PlatinVC. The algorithm takes care to distribute the snapshots among the ***maintaining peers***.

Locally applied commits are shared using the push operation (*O-9*) of PlatinVC, introduced in Section 7.4.3. In this operation the local history of the local module is copied to an accompanying ***Peer-to-Peer Module*** and pushed in a background operation to the responsible maintainers in the network. The peer-to-peer module is hidden from the user and should only be manipulated using operations of PlatinVC, as direct manipulation could break the ***global repository***'s consistency.

The push operation is demonstrated in the action diagram in Figure 45. Basically the peer whose user created new snapshots computes which peers are responsible for storing them and sends them the new snapshots. In the first step the new snapshots are examined. All folders whose artifacts were updated are collected. All maintainers responsible for those folders need to receive the new snapshots. In order to route a message to them the identifier of the folders is computed by hashing the modules name, the name of all folders on the path and the folder in which artifacts were changed, as mentioned earlier.

The activity *AD open socket for push* serves two purposes: A TCP/IP socket for the actual data transfer is opened by the maintainer and the physical address of the maintainer is identified. Following the ***DHT mechanism*** a peer can be responsible for tracking the evolution of the artifacts from more than one folder. The peer of the developer, who initiated the push operation, ***routes*** a request to each folder's identifier in parallel executed processes.

The request contains *some* ***history cache*** entries, which are needed to identify the missing snapshots on the maintaining peer. The actual algorithm which determines which history cache entries are to be sent is described in Section 8.4.6. The answer is either a socket and some entries from the maintainer's ***history cache***, needed to identify which snapshots are missing on the maintainer, or an acknowledgement, indicating that the snapshots have already been applied. Knowing which snapshots are missing a ***bundle*** containing these snapshots is created and sent using the opened socket to the maintainer. A **bundle** is a data structure created by Mercurial. It packs all new snapshots with their actual changes and all metadata in a file which is compressed. An individual bundle is created and transferred to each actual maintainer, who

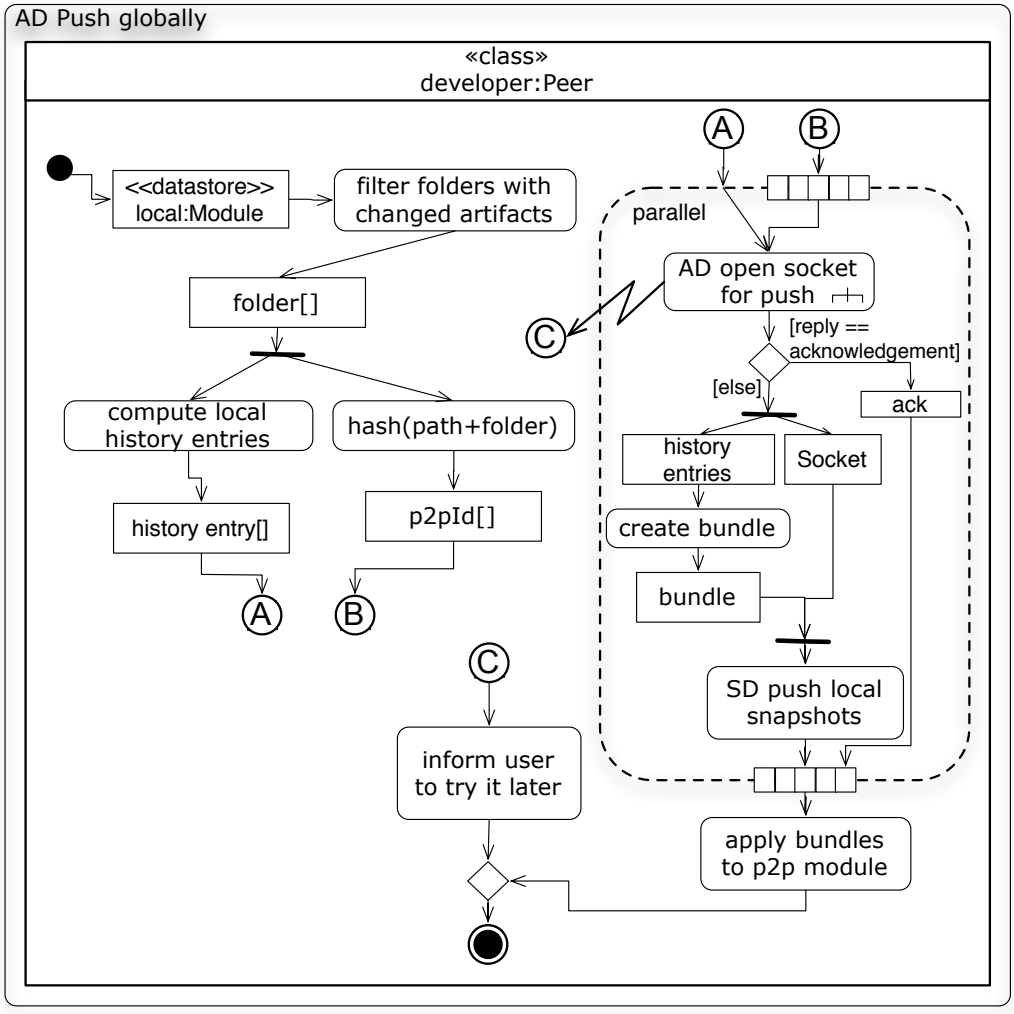


Figure 45: Activity diagram of the global push operation

confirms its successful storage and replication. Only when all contacted maintainers respond with an acknowledgement are the new snapshots applied to the peer-to-peer module on the initiator's peer, reflecting the global state. If anything goes wrong after several failed handling mechanisms (i.e. the maximum number of retries has been exceeded), the *user is informed to try it again later*.

Figure 46 shows how a *socket is opened* on the maintainer. The request is routed using the reliable routing mechanism presented in Section 8.4.4. When a peer is the maintainer of multiple folders to which new snapshots are to be pushed, it receives multiple messages, one for each folder. Only the first is answered, subsequent messages are discarded. The pushing peer does not wait for an answer to these additional messages once it received the first acknowledgement, which states for which folders the maintainer is responsible for. If the maintainer already applied the snapshots in question it resends its acknowledgement. This can happen, if a peer failed in a previous push attempt, which could have been any other maintainer or the pushing peer, before it received the acknowledgement. The maintainer computes which snapshots are missing (might be more than the newly created snapshots), opens a socket, and sends this information to the initiating peer. If the maintainer fails before it can send the answer the request will be resent after a time out. The next closest peer will now receive and progress the request.

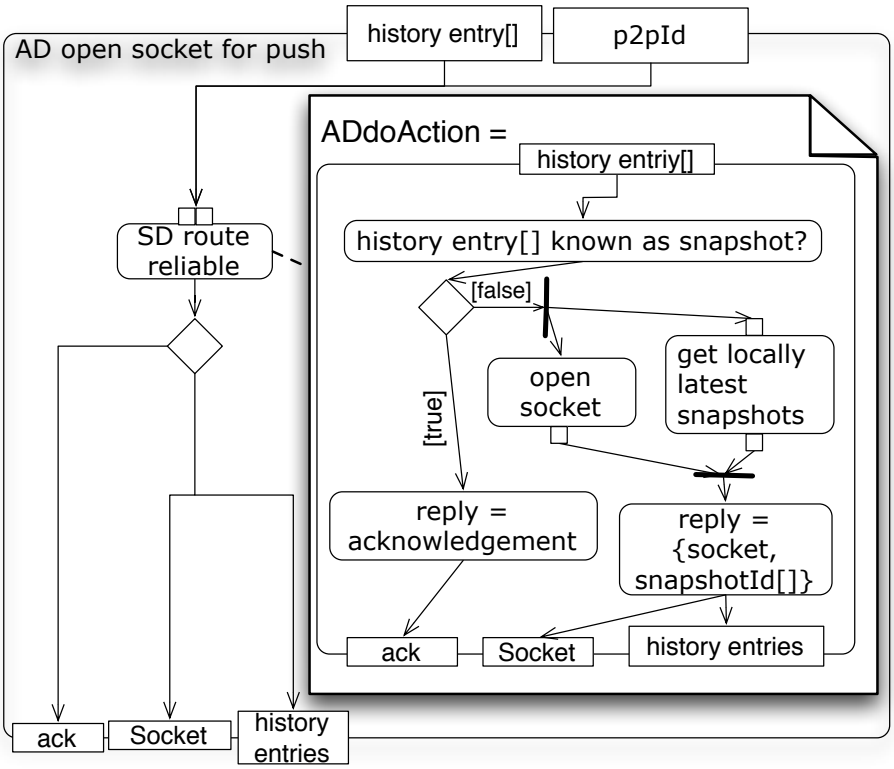


Figure 46: Activity diagram: Open socket during push globally

The actual transfer of the snapshots using the opened socket is illustrated in the sequence diagram in Figure 47. The developer's peer sends the prepared bundle using the opened TCP/IP socket. This bundle is applied to the maintainer's peer-to-peer module. Two processes are started in parallel afterwards, one sends the bundle to the replicating peers in the maintainers ***neighborhood***. These peers apply the received snapshot as well to their peer-to-peer module and update their history cache. In the other process the updated history cache entries are sent to the maintainers of the folders on the path to the folder maintained by the processing peer. Only if the acknowledgements from all peers, replicas and other maintainers, are received the snapshots were stored successfully and an acknowledgment is returned to the pushing peer.

The history cache is updated as described Figure 48. To each peer who is responsible for a folder in the path to the folder the updating peer is responsible a message, containing the new history cache entries, is routed. Upon receiving these entries the respective maintainer checks if any parent entries are missing. If entries are missing the peer asks the updating peer with a direct message. Which history cache entries are sent in this message exchange is elaborated in Section 8.4.6. Having all missing history cache entries the maintainer updates its replica peers. When it receives their acknowledgement an acknowledgement is sent to the initiating peer indicating that the process successfully terminated.

If in spite of the repeated messages any subprocess ultimately fails the user is informed to try pushing the locally recorded snapshots at a later time. If the network is unavailable for a longer period of time, but snapshots have to be shared with certain colleagues (e.g. all being on a train travel), the snapshots can be exchanged using Mercurial only (e.g. with ***bundles*** using USB flash memory drives). When the network is available later the presented mechanism can handle duplicated push operations on the same snapshots. If possible, it is preferable to build a small network and use PlatinVC to exchange snapshots. When the complete network is available later the mechanism described in Section 8.6 takes over.

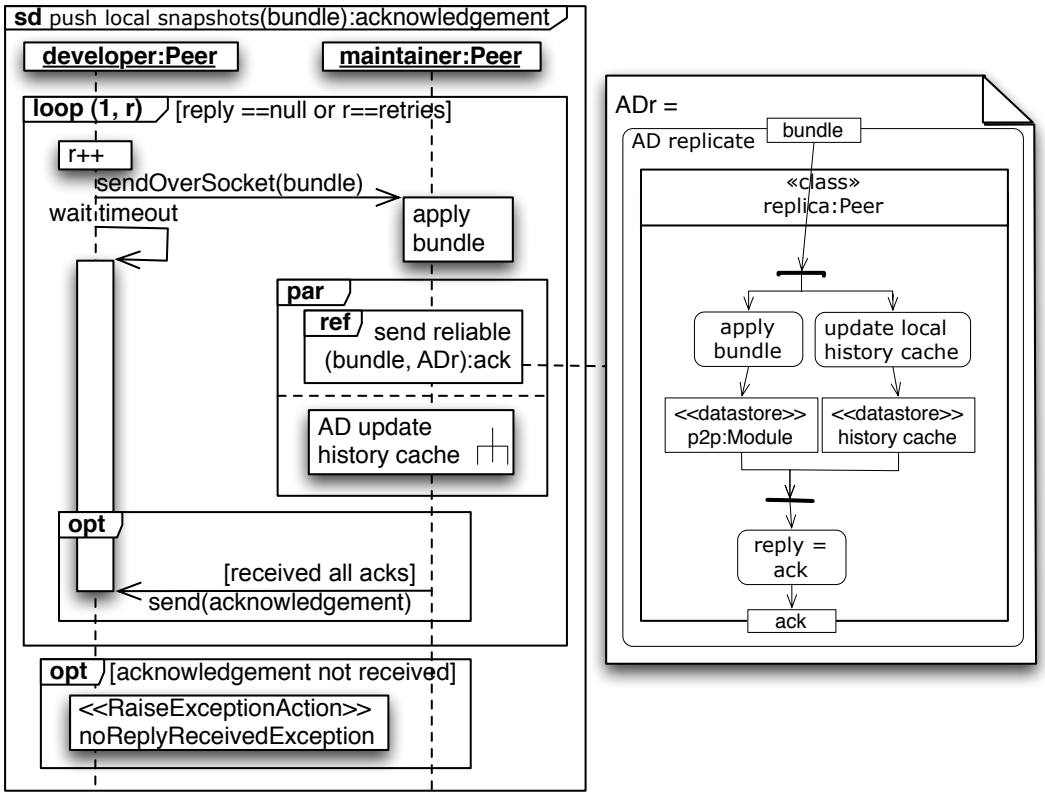


Figure 47: Sequence diagram: Push local snapshots

If a new module is created it is internally handled with the presented share mechanism. The initial snapshot is distributed among the responsible maintainer, as well as any subsequent snapshot. It does not matter if the module is not already present - it is created if it is missing. This is actually the same situation, as if a new peer joins and takes over the responsibility for a folder in a module, which is unknown to it. A peer-to-peer module is created on this peer in this situation as well.

Analysis of Design Alternatives

An obvious alternative, which would improve the time needed to execute the push operation as well as the number of messages, would be to omit the step in which the history cache on the other maintainers is updated, described in Figure 48. However, as a consequence the pull operation would be less efficient. Instead asking only one peer, who answers with the latest cache entries of all snapshots, multiple maintainers would have to be contacted. In the case, where updates of all artifacts are desired all maintainers instead of only the maintainer of the outermost folder would have to be requested. This outweighs the disadvantage of a slightly less performant push operation a lot. Additionally, a mechanism which announces new artifacts and folders would be needed.

UPDATE HISTORY CACHE When sharing or retrieving snapshots one has to update one's local ***history cache***. We cannot assume that either the requesting peer or the asked maintaining peer has a subset of the snapshots known to the other peer. All snapshots stored in the history cache of the local peer might be new and thus unknown to the maintainer as well as vice visa. So how can we synchronize the history cache on both machines with a minimum number of messages that transfer a minimum number of history entries?

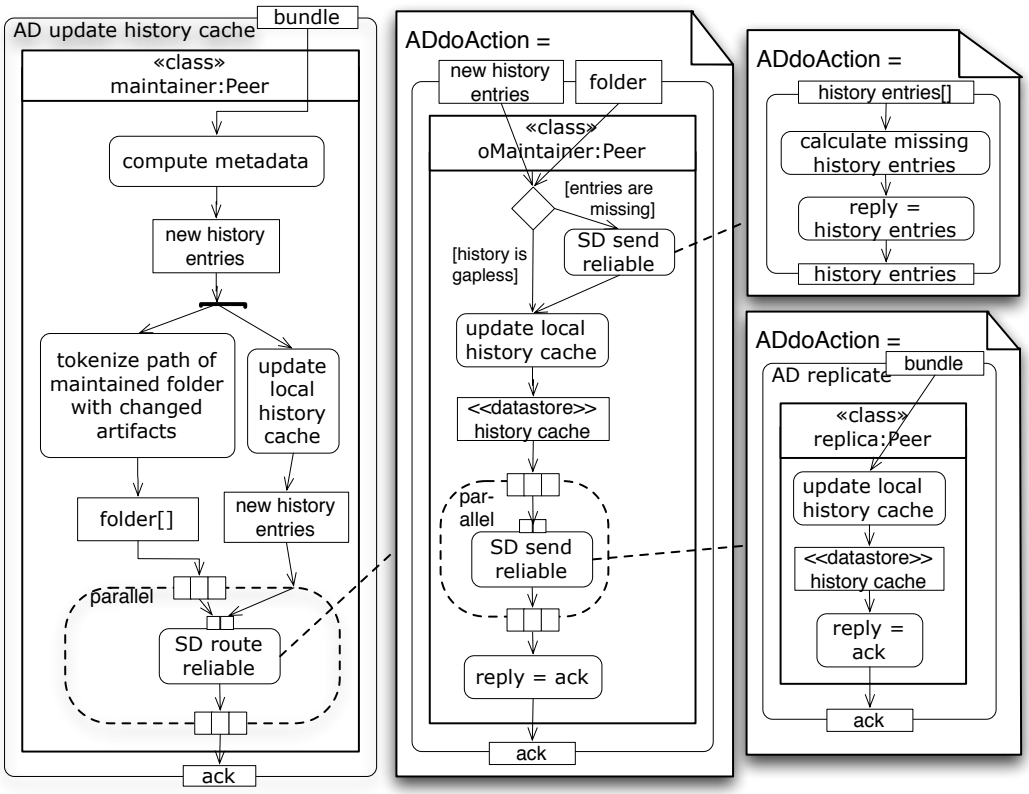


Figure 48: Activity diagram: Prepare to push new history entries

Since the history caches might contain any form of overlapping entries, ranging from no common entries to identical history caches, there is no optimal solution. Thus we implemented a heuristic solution: We know that after peers contacted each other their history caches have the same entries. In this way history entries are propagated through the network as a side effect of the operations offered by PlatinVC. But peers can disappear at any time. In an extreme case a ***nomadic developer*** might leave one network and join another, thus bringing multiple new history entries into the new network while many entries residing in the new network are missing on his peer. The older an entry is, the more likely both peers store it in their history caches. More recent entries are less likely to be shared with the other peer. When exchanging history entries in the first step of the pull operations, described in Section 8.4.5, the latest snapshots are yet to be shared.

The initiating peer first sends a subset of its local history entries. The receiving peer identifies known entries in each unmerged branch, starting from the most recent versions. All entries that represent later snapshots are sent back to the original requester. In view of the fact that PlatinVC provides at least ***causal consistency*** all earlier snapshots have to be known by the initiating peers. Of cause some entries might still not have to be transferred, but this should be a negligible number.

The subset of the initial sent history entries is deducted by applying the logarithm function to each unmerged branch. Thus, the gap between chosen entries is huge among the old entries, and small among the more recent ones. The resulting subset includes very few old and many newly created snapshots.

N.B. We cannot simply mark which entries have already be shared and which are yet to be shared. If a peer synchronizes its history cache and goes offline all other peers might disappear as well. When it joins a network again all participating peers might be new and the shared snapshots could have been lost or may never have been present. The first case is unlikely, as it would mean that a large number of peers failed in an amount of time, which was too short

to replicate stored data. The latter case is normal when a nomadic developer travels between disconnected networks.

8.4.7 *Conflict Handling*

We saw in the example given in Section 8.4.5 how conflicts are handled in PlatinVC. When snapshots are shared conflicts are ignored. Conflicts cannot be reliably detected in this phase, as a currently disconnected peer could have conflicting snapshots. If the network is partitioned, as discussed in Section 8.6, conflicting snapshots could exist in each network, which would be noticed later, when the partitions merge. A user should follow the principle to update his local module prior pushing new created snapshots. Each pushed snapshot is accepted, leading to an unnamed branch. A developer is not forced to solve conflicts, and can publish his work after a hard day in closing time. With traditional systems, which force a developer to resolve conflicts, the developer might postpone submitting his modifications to the next working day, leaving them in danger of becoming lost and unavailable to distant colleagues with different working times.

When conflicting snapshots are retrieved during the pull operation a user is informed of the conflict. She can decide to ignore the conflict which automatically creates a new branch, as discussed further in Section 7.3.1. Otherwise all conflicts can be reconciled locally before they are shared with all other developers. As detailed in Section 7.3.1 handling snapshots in this way does not lead to a lot of uncontrolled branches. Of cause a lot of uncontrolled branches could be created, but this does not happen unnoticed (like it would in a ***distributed version control system***). These branches would only reflect the uncontrolled organization of the developed project.

It can happen that developers publish their changes at the same time, after each of them pulled the latest snapshots which were not conflicting. These conflicts will be detected delayed, by any user pulling the latest snapshots afterwards.

In contrast to the conflict handling in ***centralized version control systems*** all snapshots are recorded. In those systems conflicting snapshots are rejected. A developer has to merge the latest snapshot with his working copy to be able to submit the result as a new snapshot. In PlatinVC the local snapshot should be merged with the latest snapshot as well, but all three snapshots are recorded in the shared repository.

8.4.8 *Additional mandatory mechanisms*

Moving or Renaming Artifacts

An artifact is identified by its full path rather its name only. Thus moving the artifact is equal to renaming it. Mercurial handles artifact renames in a way which fits to our repository distribution naturally. If an artifact is renamed Mercurial deletes the artifact in the old location and creates a new one in the new location.

As long as the path to the artifact did not chang nothing changes in PlatinVC. If the path changed the new snapshot is stored by another maintainer, namely the one whose identifier is close to the ***hash value*** of the renamed path. In the push process which shares the new snapshot (that includes the moved artifact) the new maintainer gets all missing snapshots - which can be the complete history in the worst case. The former maintainer does not receive any updates regarding the moved artifact anymore, except a snapshot is based on a previous snapshot, where the artifact has not been moved. The snapshot in which the move is recorded, however, is stored on the new and the old maintainer automatically. In the folder of the new maintainer the change has the form of a new artifact (which is actually the moved old artifact). The change in the old maintainer's folder is the deletion of the artifact. Thus, even when the automatic isolation feature described in Section 7.3.1 is used where a developer pulls updates relative to specified folders only, the relocation of the file will be noticed.

New Artifacts and Folders

New artifacts in a known folder do not bring any problems. They are tracked by the maintainer responsible for the folder it resides in and retrieved by any peer who pulls the latest snapshots from this maintainer.

A new folder with new artifacts is maintained by a peer whose identifier is ***close*** to the peer's identifier. If the new folder has been pushed along with changes in other folders the snapshot ID is known to the old maintainers as well. If it has been pushed as the only change the snapshot is stored by the new maintainer only. We saw that in the push operation the new history cache entry is propagated to the history cache of the maintainer that care for the folders on the path to the new folder. Whenever a peer retrieves an update during the pull operation from any of those maintainer it is informed about the new folder.

8.5 MAINTENANCE MECHANISMS

Whenever a peer leaves or joins the overlay network the ***neighborhood*** of some peers changes. The ***maintenance mechanisms*** of FreePastry notify PlatinVC about those changes. Whenever a peer fails it is replaced by its closest neighbor. From the point of view of the other peers in the neighborhood it is the same effect as if a new peer joined, as an already present peer slides into the neighborhood set.

Figure 49 and Figure 50 describe the maintenance mechanisms performed by PlatinVC. A joining peer first executes a push to ensure that snapshots missing in the network are shared again. At the same time the protocol described by the figures is performed. Any peer who is notified by FreePastry checks if the joined or failed peer was its closest neighbor having the next smaller or greater number. If this is not the case the peer does nothing. The peers who perform the maintenance protocol are the two closest neighbors of a joined peer, or the closest neighbors of a failed peer. In the latter case the two peers see each other as the newly joined peer.

If the new peer is ***closer*** to the identifier of a maintained folder it replaces the old maintainer. The active peer checks whether it itself is the maintainer of any folder, and if the new peer is closer to a maintained folder's identifier. If this is the case the peer does not forward any push or pull requests, which would be handled by the new maintainer, until it is updated. For the rest of the protocol it does not matter, if the new peer joined or was already present but became the closest neighbor as a result of a leaving peer.

As described in Section 8.4.6 the identifier of some snapshots are sent to deduce the missing snapshots. A time out is not acceptable, so the message is resent until the peer replies or FreePastry notifies the final failure of that peer. Figure 50 explains how the new peer is updated. Using the received snapshot identifiers the active peer creates a ***bundle*** of all missing snapshots and sends them to the new peer, again repeated until they are received or FreePastry notifies the final failure of the receiver. The new peer updates its peer-to-peer module as well as its history cache and is ready to serve as a replica or maintainer. If the new peer is closer to the identifier of a folder the active peer no longer intercepts any requests.

During this process the active peer could fail as well. In this case the next closest peer becomes active and updates the new peer. If a peer should become active but is not updated itself yet, it delays executing the maintenance protocol until it is updated. In this way all peers are updated in a chain reaction eventually.

8.6 FAILURE HANDLING

We already saw in Section 8.4.5 and Section 8.4.6 how faulty situations are handled. In this section we discuss a few special cases.

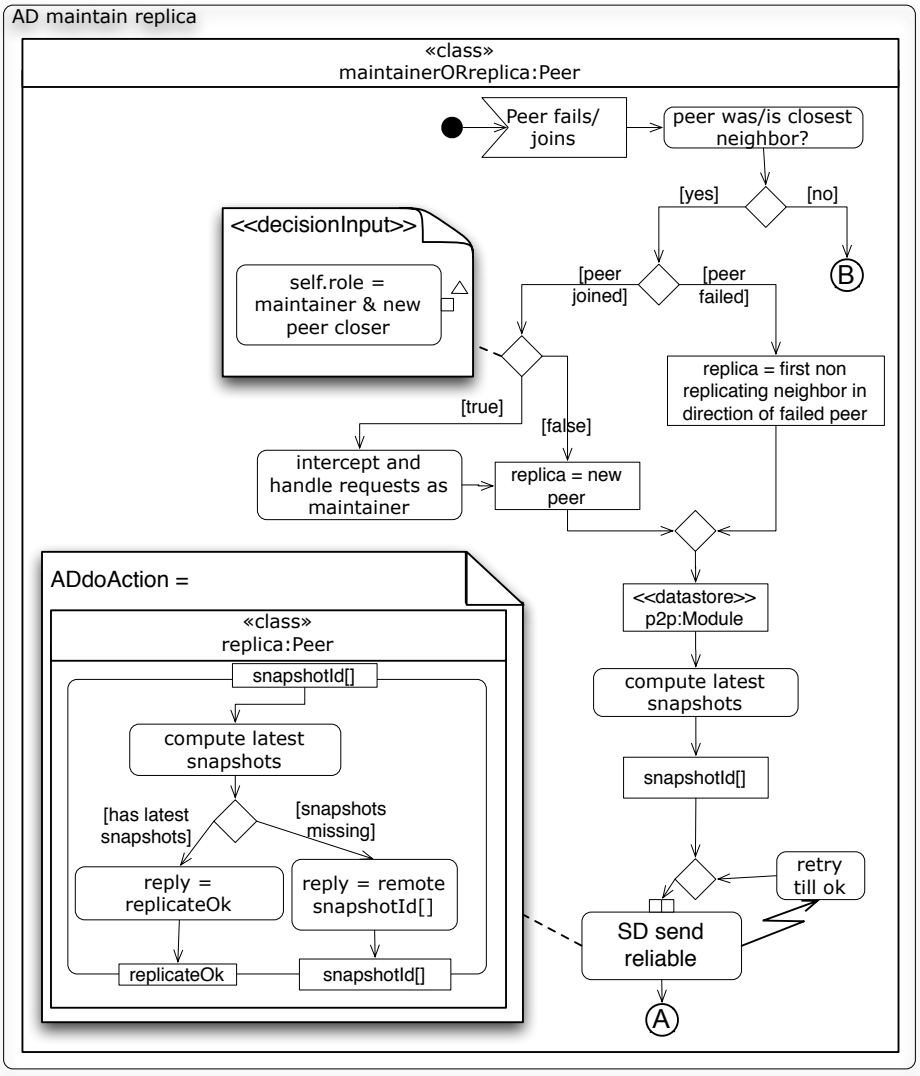


Figure 49: Activity diagram describing the actions taken when the neighborhood changes (part 1)

8.6.1 *Handling Network Partitioning*

Whenever a group of peers loses the connection to another group of peers the overlay network falls into **partitions**. A peer keeps active connections to its ***neighbors***. These neighbors are chosen by their identifier, which is given following a normal distribution. Thus it is unlikely that a large part of neighbors are disconnected as a result of normal ***churn***.

However, if the physical network connection is interrupted a group of peers could be disconnected at once. If some peers are in the subnetwork of an Internet service provider (ISP) and the remaining peers are in the subnetwork of another ISP, the connection between the ISPs could be interrupted. To name another likely example the outworld connection of a building could break down, leaving only the machines in the building connected.

The ***maintenance mechanisms*** presented in Section 8.5 take care that the missing peers are replaced and the ***routing mechanisms*** deliver all requests to the substituting maintainer in all partitions. However, complete ***development lines*** consisting of connected snapshots could be missing. If any user pushes a snapshot based on missing snapshots the missing snapshots are published as well. They have to be present on that user's machine - otherwise the user would not be able to create the new snapshot in the first place.

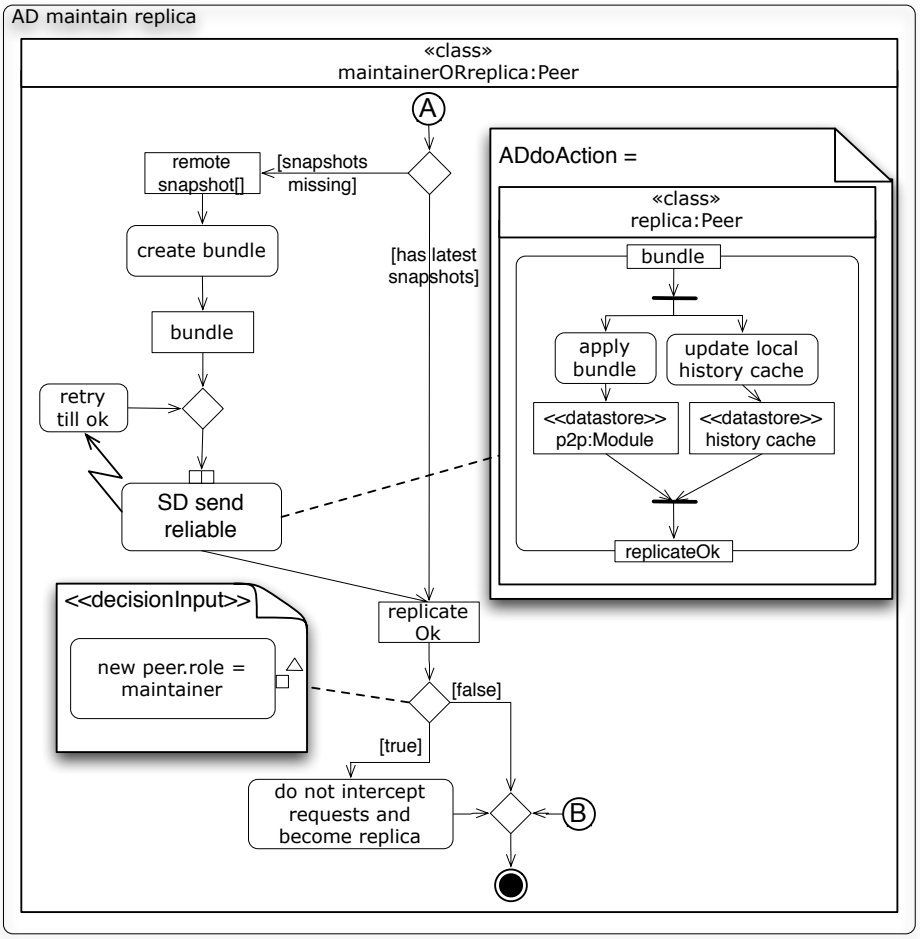


Figure 50: Activity diagram describing the actions taken when the neighborhood changes (part 2)

However, in all network partitions snapshots are created independent, which might conflict each other. The conflicts can be detected only at a later time, when the network partitions reunite. Only one maintainer among the multiple maintainers of all partitions, regarding a certain folder, is closest to that folder's identifier. PlatinVC's maintenance mechanisms ensure that the independent recorded snapshots are joined in the peer-to-peer module of the remaining maintainer.

8.6.2 *Handling Indeterministic Routing*

During extreme ***churn*** it can happen that the routing table entries of the peers are faulty. A peer might not be able to update its routing table, when an old peer rejoins and a recently entered peer fails. Another peer might still have the old entry which is correct again. Thus the paradoxical routing table entries lead to **indeterministic routing**. Depending on the intermediate peers a request addressed to the maintainer of a specific artifact could be delivered to two different peers. On a closer look this situation is very similar to the ***network partitions*** described above. The only difference is that the peers are not in disjunct sets, but rather overlapping sets. The problematic situation whereby multiple peers maintain the same folder, is the same. The previously solution is solving that problem here as well.

8.6.3 *Recovery of Missing Snapshots*

It cannot happen that a snapshot in the middle of a development line is missing - only the latest snapshots up to any snapshots can be gone. Either because they have not been shared correctly or all storing peers failed before the maintenance mechanism could copy them to replacing peers. However, whenever any peers store those snapshots, either in their local or peer-to-peer module, they are stored in the system if the peer pushes any new snapshot again. The new snapshot does not have to be based on any of the missing snapshots.

8.7 TRACEABILITY LINKS

A link can be created between artifacts, folders or modules. The link is stored in a specific file. For each linked item a companion file is created, with a generic name formed by the "." prefix to hide the file, the items name and the suffix "<-linkfile". In this file all linked items are listed. N.B. These entries express only that a link exists. No further information is stored here. The actual link document is named by concatenating the linked item's name with "<-link->" in between, prefixed by a dot as well. If a link is deleted only this link document is deleted, the inscription in the linked item's linkfile remains untouched. This preserves the information that instead of creating a new link again the old link has to be updated (i.e. undeleted). The linkfiles are stored along with the corresponding artifact, in the case of a folder in the folder and in the case of a linked module in the topmost folder. The link document is stored in a folder named ".linkdocuments". If the link document links to items which resides in the same module this folder resides in the topmost folder of the module. If items of different modules are linked the folder is stored in the topmost folder of a third module, which is named after the modules names in which the linked items are. The name is constructed by the modules names with "<-links->" in between.

Using the previously described links not only can artifact's can be traced, but different modules can be aligned as well. A practical usage would be to refer to an independent library which is used by a project. The link's metadata could specify the needed snapshot of the linked library.

8.8 SUMMARY

Rather than reinventing existing solutions, PlatinVC integrates software that are proven to be stable in many other projects, and focuses on implementing the new parts. Our solution does not depend on a concrete ***peer-to-peer overlay protocol*** and only requires a solution that provides key based routing, which is implemented by any ***structured peer-to-peer protocol***.

The modification on a user's working copy is tracked and stored using *Mercurial*, a distributed version control system (see Section 5.2.4). As a side effect, all tools that support or extend Mercurial, e.g., to visualize the recorded version ***history***, can be used with PlatinVC as well. In overcoming Mercurial's limitations, which result in a lower ***degree of consistency***, we developed s mechanism to share committed snapshots with all fellow developers. As a result PlatinVC combines the benefits a locally stored version history brings with the automatic and complete collaboration of centralized solutions. With the sophisticated distributed mechanisms presented in this chapter we were able to combine the best features of centralized and distributed version control systems while eliminating their drawbacks.

We carefully considered situations which can lead to a faulty execution and designed mechanisms that counteract those effects and maintain a highly consistent and available ***repository***.

9 PROTOTYPICAL SOFTWARE DEVELOPMENT ENVIRONMENT

After taking a deeper look at PlatinVC in the last chapter we introduce it from a visible perspective in this chapter. PlatinVC is a good supplement to any development environment. We developed our own environment, fitting for GSD, where PlatinVC is a component among others. Although PlatinVC is the most sophisticated component, the environment and a few additional components will be introduced in this chapter.

9.1 MODULAR DEVELOPMENT OF PEER-TO-PEER SYSTEMS

A modular design was important to us. It enables us to exchange the system's components with better versions or integrate new components that offer additional functionality and are able to interact with other system components.

A key decision was to enable multiple applications to use the same ***overlay network*** for communication. Today's peer-to-peer applications, such as Skype[Skyb], Joost[Joo] and Emule[KB04], all use their own implementation of a ***peer-to-peer protocol***. The protocols might by more fitting to the respective application's needs, but the overlay's resource consumption (network bandwidth, computing power, memory), caused by the periodic ***maintenance mechanisms***, stack up. If the applications complement each other and have similar needs regarding network communication, as in our case, it is more efficient if they share one overlay connection.

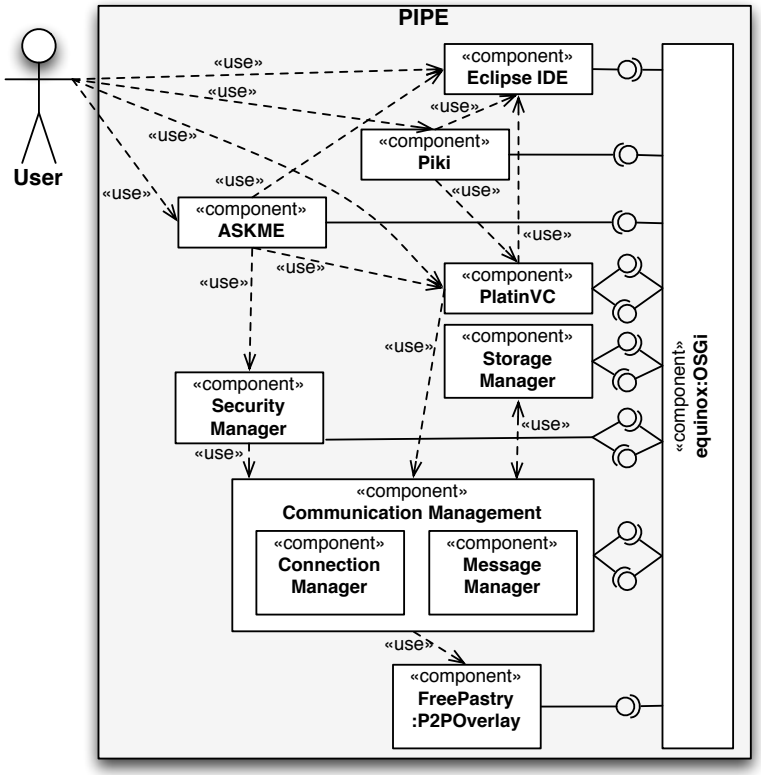


Figure 51: The simplified architecture of PIPE

We developed a framework that offers system services to components that interact with a user. Similar to the resource management framework (RMF) [FFRS02] our framework aims to decouple the complex peer-to-peer protocol algorithms from the logic of an application, to ease the latter's development.

The simplified architecture is presented in Figure 51. Our framework is called **PEER-TO-PEER BASED INTEGRATED *PROJECT* SUPPORT ENVIRONMENT** (PIPE). The spine of our system is the OSGi[All07] implementation equinox[equ]. Each component registers its offered interfaces and has access to another component's services. All messages on a machine are exchanged using this middleware. Any component can be replaced any time with a better version. Services are not offered to specific components but can be accessed by any component. In this way we do not have a classical layered architecture, where the communication of a system's component can only occur at specified borders. Nevertheless, our architecture can be separated into two parts - ***framework services***, presented in Section 7.4 and ***user level applications***, shown in Section 9.3. In RMF the network communication is abstracted to a level, where an application searches and acquires resources only. PIPE offers an application a communication interface. Otherwise implementing framework services would not be possible.

9.2 FRAMEWORK SERVICES

The component on the bottom of Figure 51 implements the ***peer-to-peer overlay mechanisms*** of the Pastry overlay network[RD01b]. The component is encapsulated by our *communication management*, which we present in Section 9.2.1. This component handles the communication for all other components. It is used by a *storage manager*, shown in Section 8.4.1, which implements the ***DHT storing mechanism***. For secure message exchange and access control we implemented the *security manager*. Currently it is used by ASKME (our communication application, detailed in Section 9.3.4) only. These four components, the communication management, the storage manager, the security manager, and the overlay implementation, are ***framework services***. The components' services are used by other components only, and cannot be used in direct interaction with a user. The communication management's services are used by the storage manager and the application ASKME. This would not be possible in a strict layered architecture, as ASKME belongs to the ***user level applications***, the storage manager to the ***framework services***.

9.2.1 *Communication Management*

This component brings an abstraction of the communication. It consists of the following three components.

Connection Manager

The connection manager handles the overlay network connection. In the connection dialog depicted in Figure 52 we can see the choices one has to establish a connection. A new overlay network can be created with *new ring*. If there is already an overlay network, which is the most common case, one can *join* by specifying the physical network address (i.e. the ip address and TCP port number) of a peer who is currently a member of that network (called the **bootstrapping peer**. *Join using known peers* allows one to join a network by trying to contact formerly known peers. To enable this convenience function we store the physical network address of the ***neighborhood peers*** we knew last time we were online. *Disconnect* in self explanatory. When a user disconnects an event is sent in the system, so that other components can react. If a component needs to execute specific actions before the connection is lost, it registers itself at the connection manager, who waits with the actual detachment from the overlay network until all registered components agree. No component can rely on this service, of course, as a machine might fail any time. Thus a user is able to cancel this graceful disconnection attempt and interrupt the connection immediately.

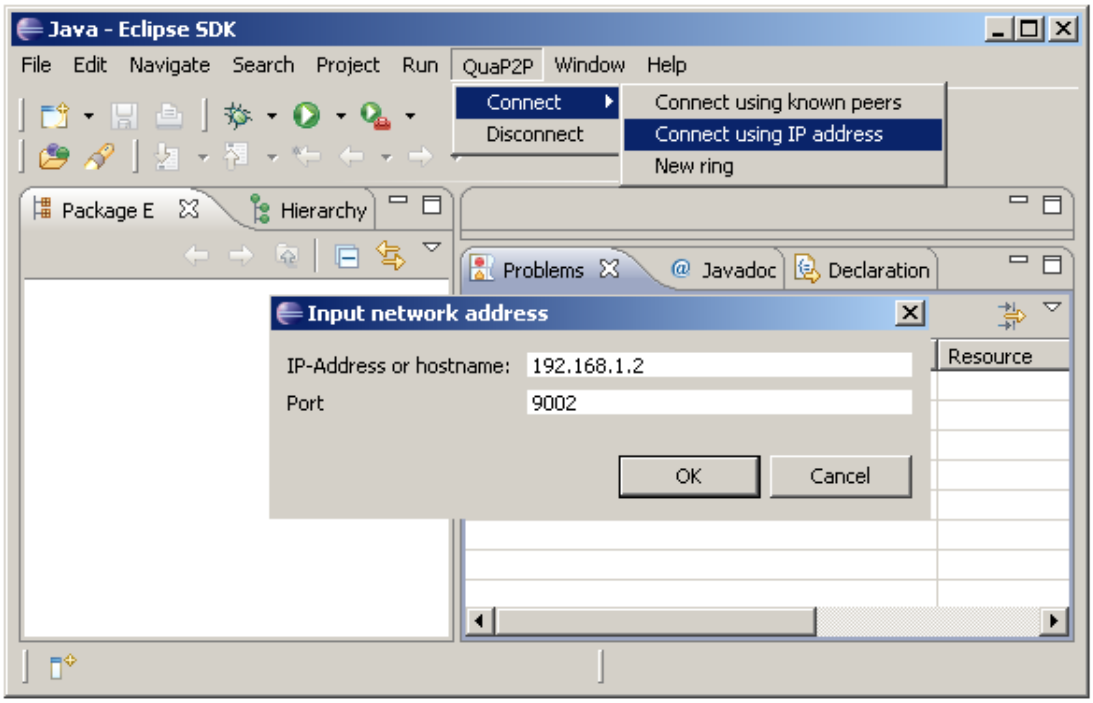


Figure 52: The connection options offered by the connection manager (running in Windows)

In Figure 53 we see the settings the system needs in order to establish a connection. The *security settings* are needed by the *security manager* only and can be deactivated. By calculating the ***hash value*** of a developer's name, the identifier of her peer is computed. In this way a developer can log in from any machine. If she logs in from another machine, all versions stored under the developer's peer's identifier have to be transferred to the new machine - which is an automatic process carried out by PlatinVC's maintenance mechanism. However, when modules of large size have to be transferred over the network we recommend transferring them using physical data storages such as USB flash memory drives or DVDs. If a required module exists as a user module, PlatinVC takes the required versions from the local machine.

Message Manager

Any message in Pipe is received by the message manager. By examining the message type it decides to which component the message has to be delivered. In addition to this basic function it offers basic message implementation that can be extended by any other component. These basic implementations include any-, multi-, and broadcast messages, as well as *offline messages*. We introduced offline messages in [MKS08]. These messages are stored in the overlay network, by the storage manager, until the destination of the message rejoins the overlay network.

9.2.2 *Security*

We developed a decentralized security mechanism for secure message exchange and access control in [Que09], which has yet to be tailored to PlatinVC. We briefly describe this system here. A more comprehensive description can be found in [Que09] and [GMM+09].

Figure 53 provides a glimpse of the security implementation in Pipe. Besides fields for providing the needed security key files a role and a company has to be stated.

Our approach is based on two components: Authentication and access control.

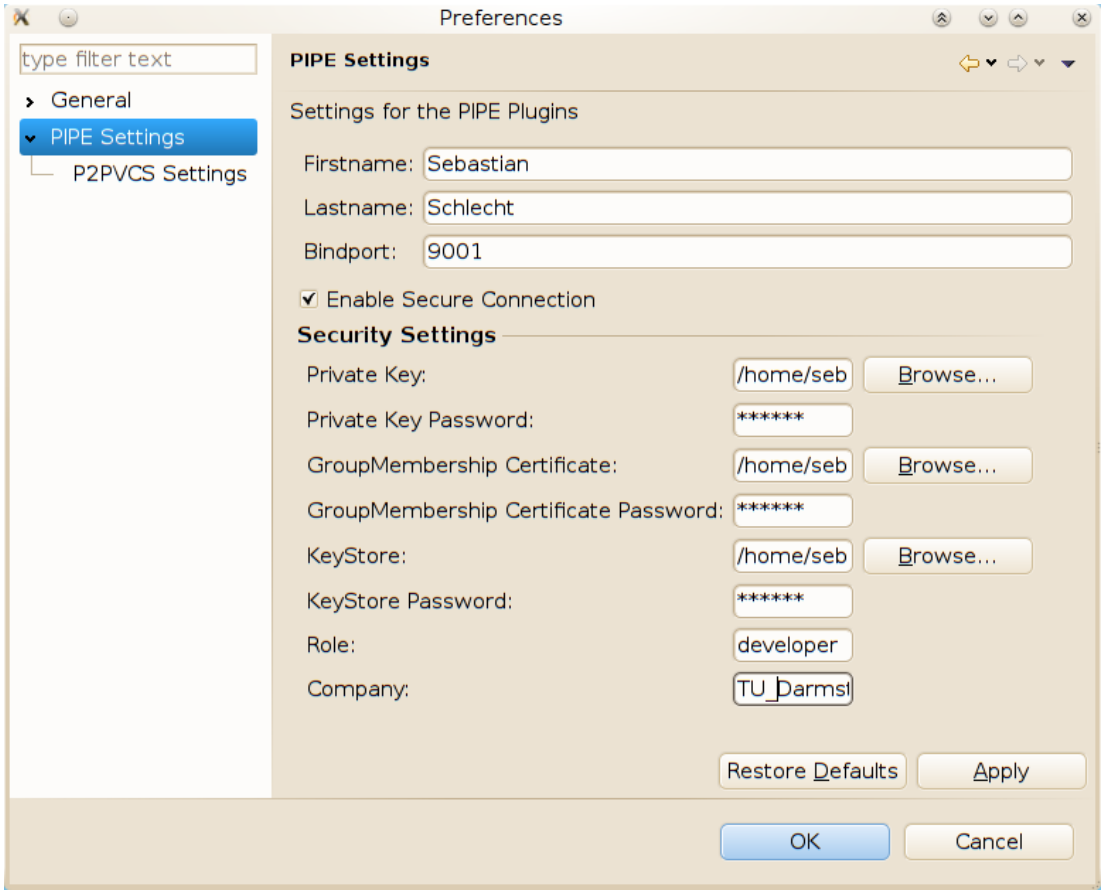


Figure 53: The settings needed by the connection manager, with security enabled (running in Linux)

Authentication

Utilizing an asymmetric key pair, as introduced by [DH79], we sign, encrypt and decrypt messages and content in our system. Rather than calculating a peer's identifier by applying a ***hash function*** to a user's name, we use the public key of a user as a user's ***peer ID***. In this way the pseudonym identity of a user is bound to a public key, which blockades a man-in-the-middle attack [Den84]. To avoid spoofing of identities we use a challenge-response mechanism upon joining the network: The ***bootstrapping peer*** sends an encrypted random number. Only if the receiver can decrypt it and sends back the correct number, the join process starts, in which the joining peer obtains its initial contacts.

Using the public key as the peer ID can be implemented in two ways, which we both implemented. The peer ID can be extended to be as long as a public key. The hash function has to be replaced so that all IDs in the network are calculated using the extended hash function. When extending the hash function is undesired the peer ID can be calculated based on the public key of a user. A limitation is that a user's public key has to be sent to other users, who can only verify it, if the peer ID is known and trusted. To avoid a man-in-the-middle attack the solution in [SSF08] could be used, which enables a decentralized key exchange.

Trust is based on our modification of a web-of-trust. Usually development projects are hierarchically ordered. Even open source projects have a project founder, although the remaining hierarchy tends to be flatter in comparison to the structure of a commercial project. This single person on the top of a project's hierarchy is the **root of trust**. It can certify another user's credentials. A certified user can certify other users, etc. In this way, our system does not depend on a central certificate authority. A user's credentials can have any attributes, such as a user's real name or role, i.e., security level, in a project.

Access Control

Using a user's credentials we developed a role based access control mechanism. An artifact does not have an access control list, where authorized users have to be entered. An artifact has attributes, e.g., its status (draft, final, ...), security level (protected, confidential, ...), etc. A user's access rights can be looked up in an extendible policy list using the user's attributes and an artifact's attributes. This policy list is stored in the system and can be read by any participant, but only written by authorized persons, starting with the ***root of trust***, a special user who can allow other persons access. Although we have concurrency control, it is important to take care that no contradictory latest versions exist. There might be multiple policy lists or multiply parallel versions (in different branches), but all of them must form a contradiction free list. Thus only a small number of persons should be eligible to modify this list.

The policy list is enforced by any user who has access to the artifact in question. The idea is that this user can be trusted, as he could access the artifact and hand it over anyway. It can be retrieved encrypted by any participant. It is encrypted using a symmetric key. Any peer who has read access can have this symmetric key. Initially this key is distributed among a specified number of peers, whose users have read access. Along with the artifact, an incomplete list of authorized peers is stored. A peer who requests an artifact for read access retrieves the encrypted artifact and this list, and contacts any peer from that list, sending its attributes. This peer retrieves the latest version of the policy list. Having the requestors credentials, the artifacts attributes, and the policy list, the peer can determine the access.

Write access is controlled by all storing and reading peers. Whenever an artifact is changed the author signs the artifact with its private key. Upon storing an artifact, the storing peer checks the policy list for write permission. Any peer, who later accesses the artifact for reading, checks the permission again, as the storing peer could work together with an attacker. Again, only authorized peers can manage the access control.

Our security mechanism features a blacklisting mechanism that detects malicious collaborators as well. If access has been given to a non authorized participant the issuing peer might be blacklisted. If it is blacklisted, all certificates it issued become invalid. The peer who issued the certificate to the blacklisted peer might be blacklisted as well. If a peer tried to gain access control to an artifact it does not have access to, it can be blacklisted as well. A blacklisted peer is excluded from the overlay network and prevented from rejoining. Whenever a peer is suspected to be malicious it is assigned a blacklist-point, kept in a list appended to the policy list. When a specified threshold is passed it is blacklisted. By logging the steps executed to prove malicious behavior, we take care that malicious peers cannot blacklist innocent peers.

If the ***root of trust*** is malicious the entire system does not work. If a user certifies other users carelessly he might end up being blacklisted. The time needed to detect malicious behavior of a group of peers (or an attacker with a sybil attack, i.e., one user controlling multiple peers) depends on the threshold value mentioned before. But without the cooperation of a user with write access (holding the encryption key) no artifact can be read. It is possible to create new versions without permission, if the maintaining peer allows to do so. However, these unauthorized written versions can be reverted to the latest authorized version once the malicious behavior has been detected.

Access Control in PlatinVC

A storing peer, as a maintainer or replica, is assigned randomly. As it can be a malicious peer, the storing peer itself should not be able to read the stored artifacts, if unauthorized. Thus the artifacts should be stored encrypted. But to be able to use efficient ***delta compression*** the artifacts have to be readable. When access control is desired, artifacts have to be version controlled using their encrypted form and binary deltas. Alternatively the artifacts could be stored in encrypted ***bundles***, one for each snapshot, so that multiple snapshots can be sent in the form of consecutive encrypted bundles.

If a malicious peer rejects to store or provide a maintained artifact it gets blacklisted as described in the above section and replaced by another peer.

9.2.3 *Storage*

The storage manager component capsulates different storage implementations. Currently we implemented the ***DHT mechanism*** only. We needed this storage form in an earlier version of PlatinVC, where we controlled the evolution of single artifacts only (and did not support snapshot version control). Currently it is used by the message manager only to store offline messages. All stored artifacts in PlatinVC are stored locally using Mercurial's mechanisms.

9.3 USER LEVEL APPLICATIONS

The components depicted in the upper half of Figure 51 are ***user level applications***. These components directly interact with a user's input, but not exclusively. Like all components their services can be used by other components as well. An example for this is visible in the relationships of the component *PlatinVC* . It is not only used by a user but by the component *piki* as well. All user level applications consist of two, sometimes three subsystems: The core component, and a graphical user interface (GUI), integrated in Eclipse, stand alone or both. The 'use' relationship of all user level applications to Eclipse shown in Figure 51 delineate the utilization of Eclipse's GUI framework. The four ***user level applications*** are presented in Section 9.3.1 and the following sections.

9.3.1 *Eclipse IDE*

There are many tools needed in an integrated developing environment (IDE), mainly an editor to write source code and a compiler. In all modern IDEs there are, however, multiple additional tools integrated, e.g., a debugging tool that allows step wise code execution. None of these tools need to communicate with another machine. To have this essential basic support we chose to integrate the most used tool in industry and private projects: Eclipse IDE[Fou]. Eclipse itself is based on the OSGi implementation Equinox. To see it from a different point

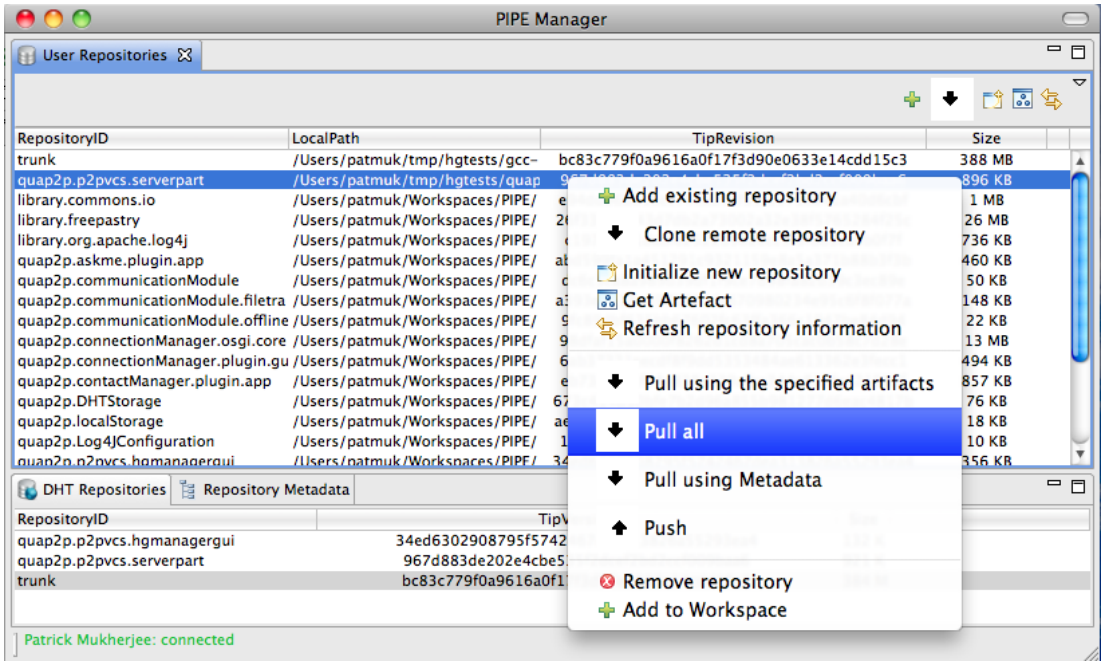


Figure 54: PlatinVC as a stand alone application

of view we integrated our components into Eclipse. Nevertheless, none of our components depend on Eclipse and can run using any stand alone GUI implementation. As Eclipse and all integrated or extendible tools are not contributed by our work we would like to refer the reader to Eclipse's documentation ([Fou]).

9.3.2 *PlatinVC- a Peer-to-Peer based version control system*

PlatinVC is the most sophisticated component at the stage, where most of our development has been carried out. It is seamlessly integrated into Eclipse as shown by Figure 58 but can be used as a stand alone product, as shown in Figure 54 as well. PlatinVC solely depends on the communication management. Earlier versions also needed the storage mechanism, but since we based our version control mechanisms on the third party product Mercurial we store all artifacts locally with the operations it offers.

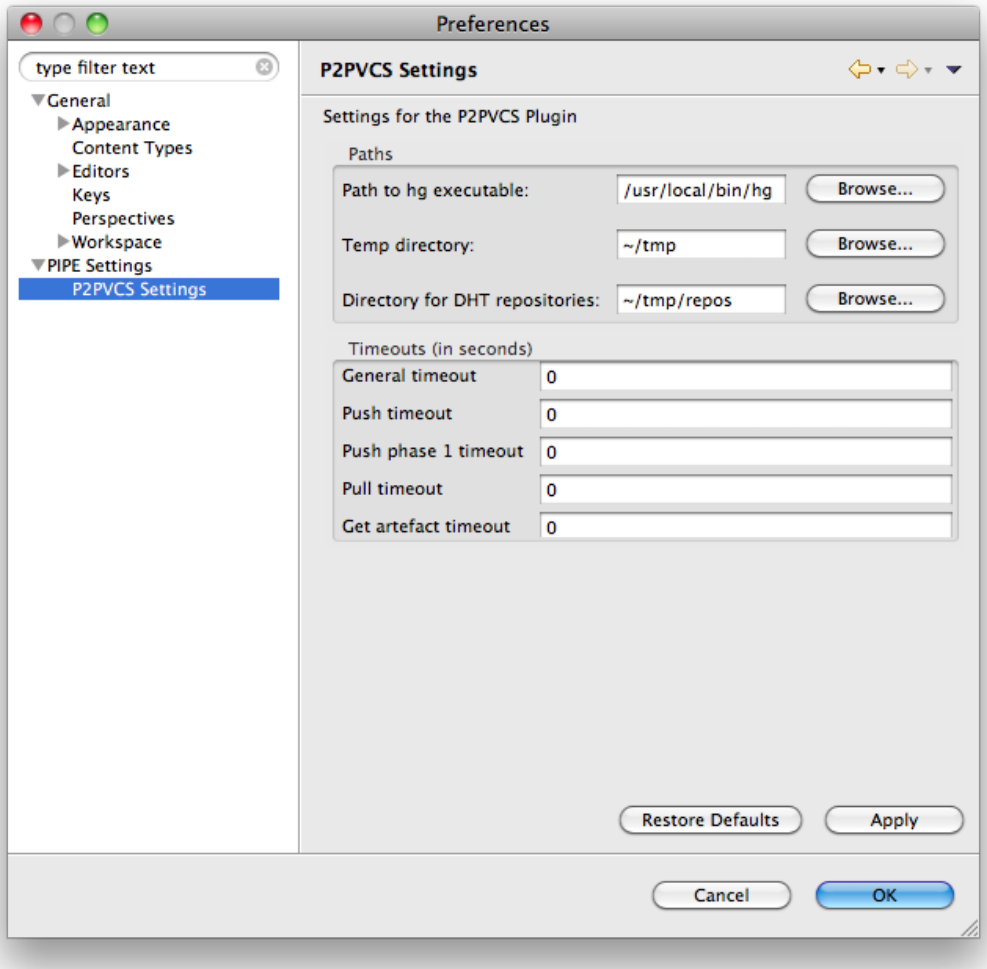


Figure 55: Settings for PlatinVC (running in OS X)

The stand alone version of PlatinVC has a smaller footprint in startup time and memory consumption than the Eclipse integrated version[1]. If a developer uses a development environment other than Eclipse, it is preferable to run the stand alone version. Unlike other version control

1 Measured on an 2.2 GHz Intel Core 2 Duo MacBook running OS X 10.6.3:
Eclipse integrated: 11 seconds startup time, 155 RAM consumption
stand alone: 6 seconds startup time, 121 RAM consumption

tools and alike all peer-to-peer based applications, PlatinVC has to be running continuously on a developer's machine, not just when the developer needs its services. Otherwise the developers machine cannot offer its services to other machines, which is the fundamental principle of peer-to-peer based applications. Having PlatinVC integrated in Eclipse when using Eclipse ensures that a developer cannot forget to start it. Providing a less resource consuming stand alone version gives initiative to let it run in the background to those developers, who do not use Eclipse. As detailed before, PlatinVC would also work if developers only turned it on when they needed it and shut it down afterwards. However, the greater the number of developers who do so, the less efficient and stable PlatinVC becomes. The actual consistency might drop to ***causal consistency*** with some versions being unretrievable in the worst case.

When running PlatinVC, a few settings, shown in Figure 55, can provide a better experience. If none of those settings are set, default values are taken. The default value for the time-outs is set to 60 seconds, but should be fine tuned for the network environment PlatinVC is operating in.

9.3.3 *Piki - a Peer-to-Peer based Wiki Engine*

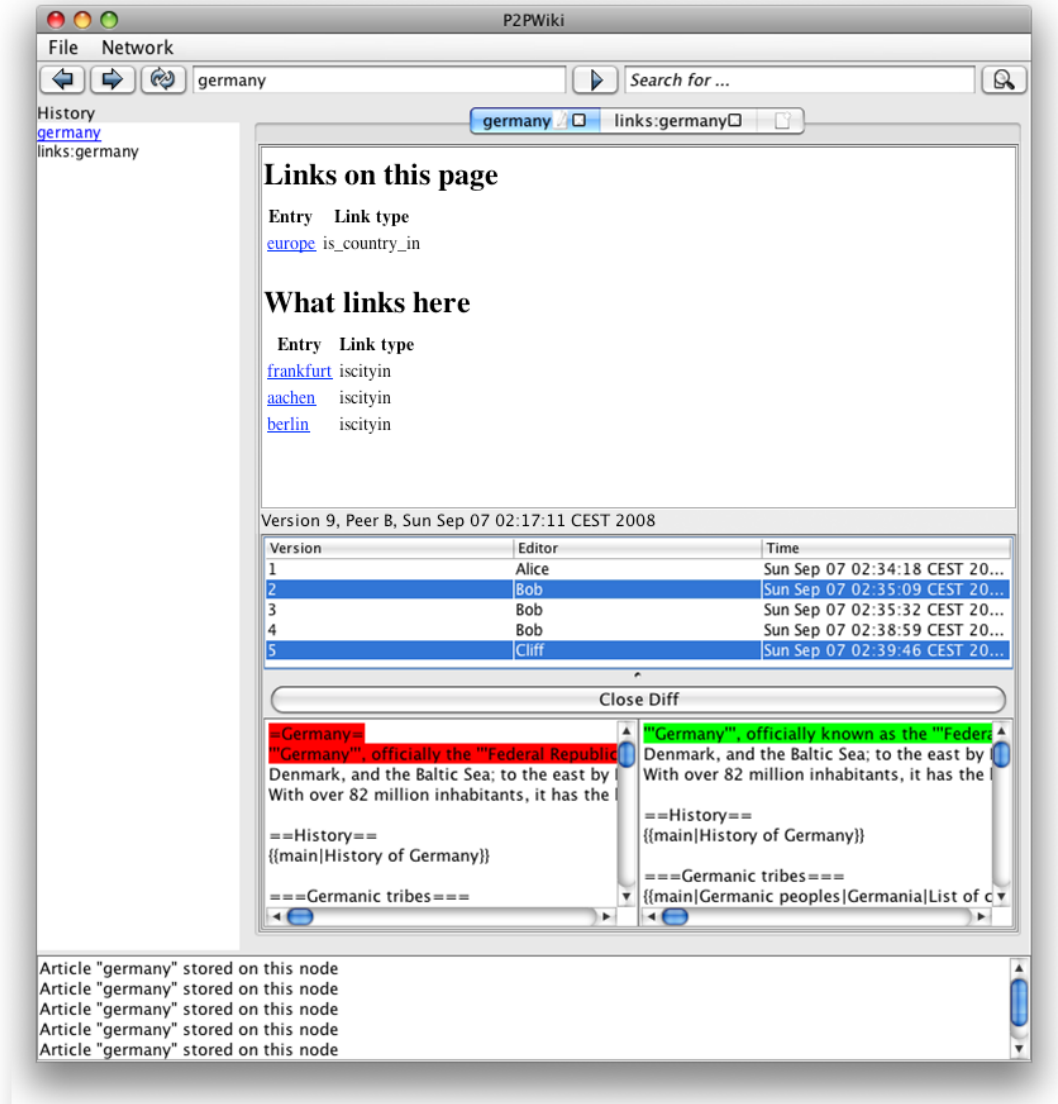


Figure 56: Piki as a stand alone application (running in OS X)

Piki is our implementation of a wiki engine that uses our version control system as presented in Figure 51. As any other component, Piki can be run as a stand alone application, a screenshot is shown in Figure 56. Beside using it as a wiki engine (see Section 2.2), it can be used for global software development (see Section 2.3) as well. In project development sharing expertise and knowledge are often crucial in order to fulfill certain sub-tasks (compare to [HM02, HPB05]). Piki can be used as a general knowledge management application (or knowledge database) or more specifically as a requirements engineering tool, as proposed by [GHHS07b, GHHR07b].

Figure 57 shows Piki when it is running as a view integrated in the Eclipse IDE. The features offered by the other components can be used here, such as the *version history* or the *diff view*, which show the difference between any two versions. Having a full blown version control system allows one to create ***variants*** of articles as well as handling editing conflicts.

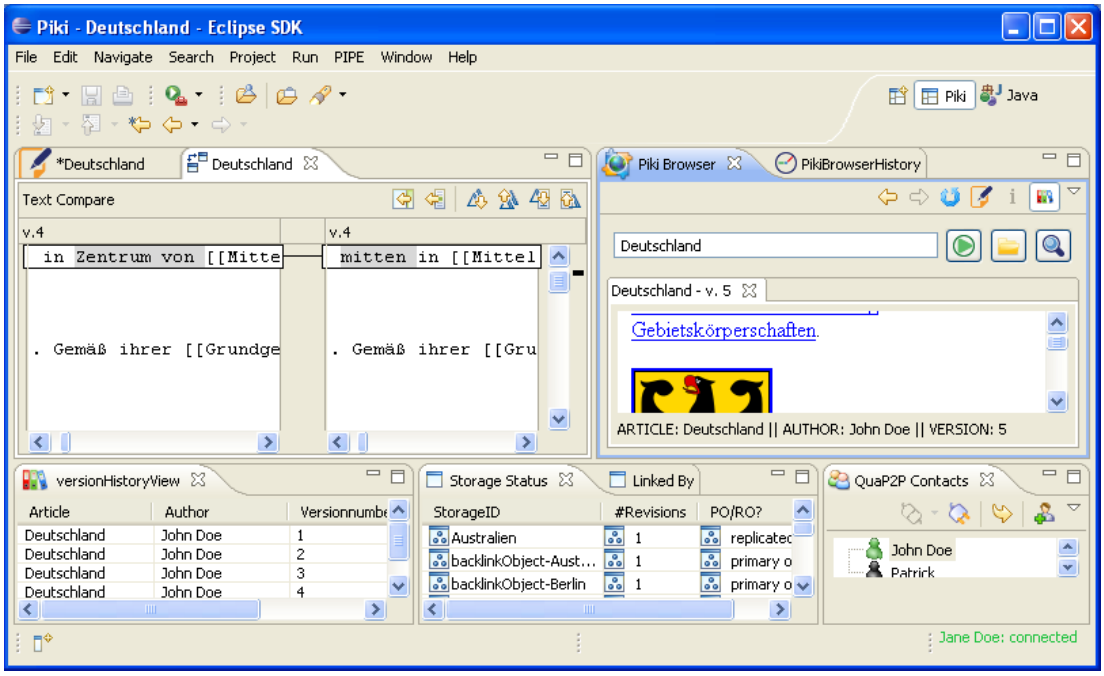


Figure 57: Piki integrated in Eclipse (running in Windows)

9.3.4 ASKME- *Peer-to-Peer based Aware Communication*

Awareness **S**upport **K**eeping **M**essages **E**nvironment (ASKME) is our tool that complements a global software development environment with communication facilities. Several investigations of distributed projects ([Šmi06, HM03, HPB05, Sou01]) pointed out that communication is one of the most critical factors for success. Communication should take place directly between participants (i.e. without an intermediate like a project manager).

We developed ASKME as a replacement for e-mail and instant messaging during project work. A screenshot showing ASKME integrated in Eclipse and interacting with PlatinVC can be found in Figure 58. In the view, in the upper right corner the contact list is shown, the messaging window is visible at the bottom of the window. ASKME automatically highlights the author of the currently visible artifact in the editor. In the example presented in Figure 58 the highlighted author is *Sebastian*. Our intention was to enable *floor conversation*: When working on an artifact meeting its last author on the floor or during a coffee break reminds a developer that he can ask this author questions. When working in physically separated locations highlighting the last author should enable the same psychological effect.

A message can be sent to any person in the contact list. A message is delivered immediately. If the communication partner is currently unavailable the message is stored by the peers in his ***neighborhood***, so that the original recipient receives the message once he is online -

independent of the sender's online status. We called this behavior, which is a mixture of classical instant messaging and asynchronous e-mail message exchange **offline messaging**.

Both mechanisms, *highlighting the last author* and ***offline messaging*** are novel approaches and were presented in [MKS08].

9.4 SUMMARY

We saw that PlatinVC is only a part of the project support environment PIPE in Figure 51. PIPE is a modular framework that capsulates services in components. The standardized inter-component communication is handled by the OSGi implementation *Equinox*. This lightweight implementation allows the exchange of any component while the system is running and can combine components written in any compatible language.

In addition to PlatinVC, which is the most elaborated component, we developed a component that handles synchronous and asynchronous communication and brings some awareness of the coworkers related contributions (ASKME). Another major component is the security manager, which handles secure message exchange and provides a role based access control mechanism.

The prototypical framework presented in this chapter can be extended with various functionalities on a plug-able component base.

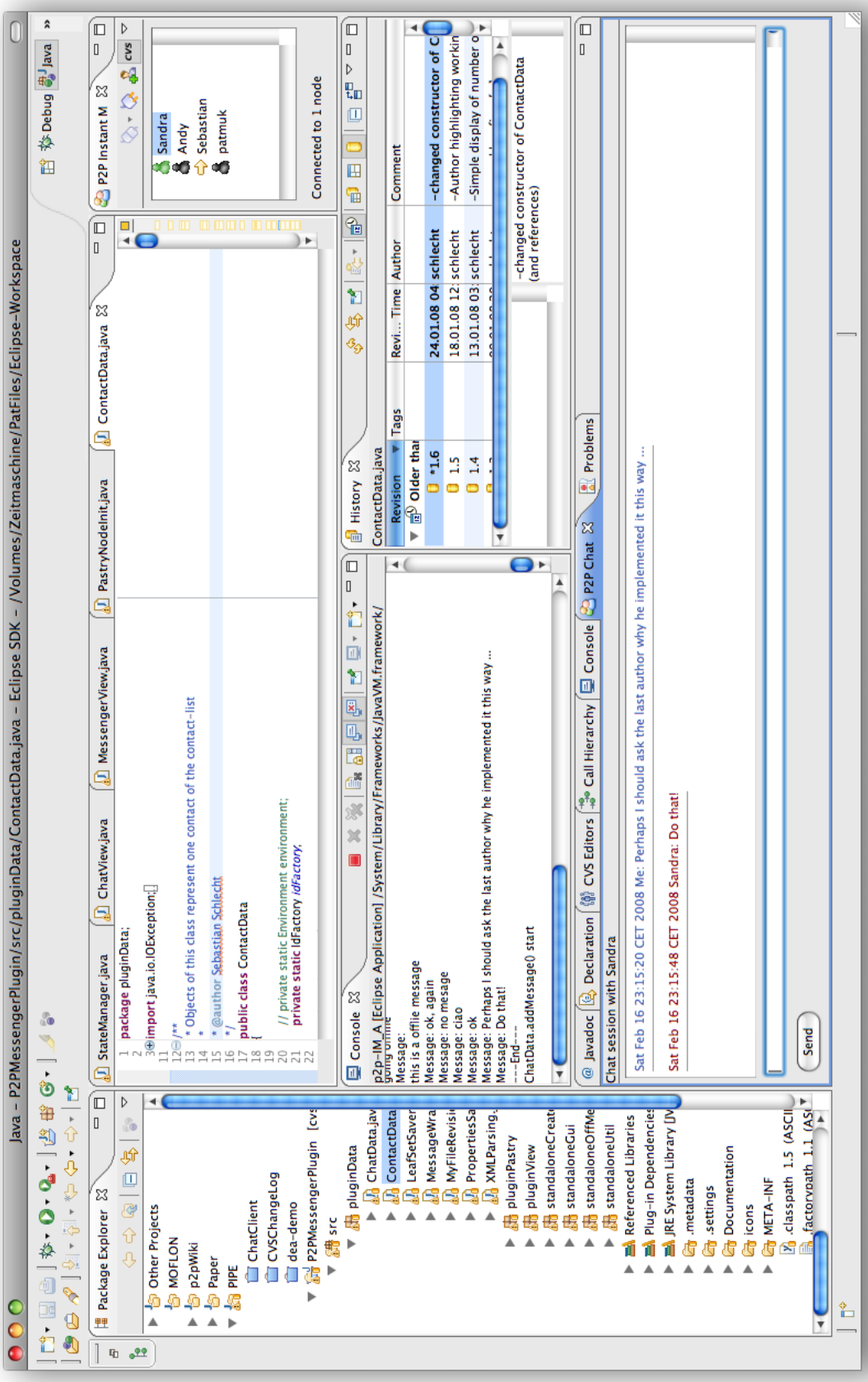


Figure 58: PlatinVC and Askme integrated in Eclipse (running in OS X)

10 EVALUATION

Following the previous chapter, in which our system, PlatinVC, was elaborated in detail, this chapter presents its evaluation, organized into six sections: specification of evaluation goals (Section 10.1), choice of evaluation methodology (Section 10.2), choise of workloads (Section 10.3), discussion of the evaluation results (Section 10.4) and comparison to the performance of the related systems (Section 10.5).

10.1 EVALUATION GOALS

The main goal of the evaluation is to examine whether the ***quality*** of the proposed system corresponds to the set of given requirements. In Chapter 3 we discuss the functional and non-functional requirements and security aspects. In our evaluation we will focus on non-functional requirements, especially consistency (described in details in Section 4.4), scalability, robustness and fault tolerance. Our extensive tests showed that functional requirements are fulfilled, as all features function as specified. A comparison of the features offered is presented in Table 5.

We quantified the non-functional requirements using the appropriate metrics. This mapping between quality aspects and metrics is described in the following subsections.

10.1.1 *Quality Aspects*

A **quality aspect** describes how well a system performs under a specific workload. A quality aspect is like an invariant - it is always present. However, under specific workloads, they more clear visible and those workloads are chosen to be a test for the corresponding quality aspects. Examples of such workloads are when many machines fail simultaneously which demonstrates robustness. All workloads we use in evaluation of PlatinVC we list and explain in Section 10.3.

We evaluated the following:

- ***quality aspects***
  - the ***degree*** of the achieved ***consistency***
  - the ***freshness*** of retrieved updates
  - the ***load*** of the system introduced by the system operations (push and pulls)
  - the ***scalability*** of the system
  - the ***robustness*** of the system
  - the *performance* of the system
- and system operations:
  - *O-9 push globally*
  - *O-6 pull globally*
  - *O-5 pull globally using specified folders*

A quality aspect cannot be quantified directly, but with set of ***metrics***.

Metric

A metric is the measurable behavior of a system, e.g., the number of messages occurring in a specified time span. A metric itself or a mathematical combination of more metrics quantify one or more ***quality aspects***. Most metrics show a trend in a system rather than giving a definitive statement regarding a corresponding quality aspect. A comparison of metric values between different solutions can describe a quality aspect.

A well known example to measure the size of software is the metric *lines of code*. By comparing this value in various codes, quality aspects like *maintainability* of software can be quantified.

10.1.2 *Metrics*

For our evaluations of PlatinVC the following mappings between observed quality aspects and ***metrics*** are used:

Consistency Degree: is measured by surveying which snapshots are pulled after a specific snapshot was retrieved the first time. Following the definition in Section 4.4, once a snapshot was pulled no older snapshot can be pulled by a subsequent pull request, otherwise only ***eventual consistency*** would be provided.

If there are more than one latest snapshot in any moment (like in the case of ***branches***) all those snapshots have to be retrieved to guarantee ***sequential consistency***. If only some of them are retrieved, the consistency degree is ***causal consistency***. N.B. A resultless request is considered to have failed and does not influence the provided consistency degree.

Freshness: describes the time that passes until a freshly pushed snapshot is retrievable by any peer. We quantified this quality aspect by measuring the time between the start of a successful push operation and the start of a pull operation that retrieved the corresponding snapshot. It turned out to be difficult to measure in our experiments, as it was hard to know when a pull request should be started, and with which frequency subsequent pull request should be executed. If the first pull request already retrieved the just pushed snapshot, it was executed to late and the measured timespan could have been smaller, if we had executed the pull operation earlier. This leads to a less well measured value for the freshness metric. If we start the pull requests to early, non could retrieve the just pushed changes. Moreover, the more pull requests we execute in a short time period, the more we stress the system.

System Load: can be measured using various ***metrics***. The size and sum of the transferred messages gives a clue of the overhead or load of PlatinVC. The number of messages alone does not quantify the load of the system in a meaningful way, as small messages are transmitted quickly and do not overload the system as much as the large messages, which use up the bandwidth of the system's participants.

Scalability and Robustness: are both *inherent* quality aspects that are especially important during changes of the network size and dynamics of a system (e.g. failures) respectively. These quality aspects are measured by comparing the systems performance under different scenarios; Different network sizes to quantify the scalability and different user participation (churn behavior) to quantify the robustness. How much different performance indicators vary give a clue about the system's scalability and robustness. The performance indicators are, for example, the percentage of successfully completed operations and the time the operations needed to complete.

Performance: is measured by the time needed to execute the operations of PlatinVC. We evaluated the prototype of our system, PlatinVC, from a user centric point of view. Therefore we measure the completion time of an operation (push or pull) from the moment a

user initiates the operation to the moment the user receives an acknowledgement that the operation has been successfully completed. However, a version is available earlier (for another user), i.e., from the moment it is stored in the ***peer-to-peer module*** of the maintaining peer. As we observe this operation form a user perspective, we do not regard it completed until a user receives the final acknowledgement.

10.2 EVALUATION METHODOLOGY

A system evaluation in computer science can be done analytically, through simulation, testbed or using real systems with real users. Those evaluation methods are listed according to the accuracy of the results: from the least to the most accurate.

The *analytical evaluation method* is the cheapest in terms of hardware and man power costs. It relies on the mathematical models of system, user, and workload that provides a prediction of the system behavior based on the input parameters. However, complex and highly dynamic systems, such as peer-to-peer systems, can only be analyzed when many important details and influencing factors are heavily abstracted in those models. Therefore many effects can be hidden in the evaluation results. The evaluation process itself is, however, fast and easy once a model is developed. We did not use the analytical method for evaluation of PlatinVC.

The *simulative evaluation method* is more accurate, as the models are not mathematically expressed but through software. It allows for more detailed models, which much more accurately simulate the real systems and its environment (workload, users, underlying network). Those models are integrated in a *simulator* which provides more flexibility in the experiments' setup, e.g., changes of user behavior, more complex experiment timelines etc. A strong point of the simulative approach is the scalability - thousands of machines can be simulated in software without the need to deploy the actual hardware. A survey [ISHGH07] among 744 IT professionals who work in ***GSD*** projects indicates that the number of participants is much less than a hundred people (a more detailed conclusion can be found in Section 10.3). Focusing on the ***GSD*** scenario (presented in Section 2.3) we do not need the ability to evaluate thousands of peers. In the wiki scenario (presented in Section 2.2) there are certainly more participants, but the expected quality is lesser than in the GSD scenario. Therefore, the benefit of simulation as an evaluation methodology is not useful for the evaluation of PlatinVC. Additionally, the simulative approach still abstracts a real system and is thus not as accurate as evaluations of a running system. During the development of our prototype, however, we used the peer-to-peer simulation framework PeerfactSim.KOM [KKM+07] to evaluate single mechanisms, such as different commit protocolls. In this way we compared the impact of different design decisions.

The most accurate evaluation method is measuring the actual quality of *a real system with real users*. This method is the most time consuming and needs the most resources with regard to hardware and man power. As the previous chapter, Chapter 9, described, we developed a real prototype of PlatinVC. All developers of PlatinVC used it for version control; However, this is not enough to say we had a real users in our evaluations. Instead, we modeled user behavior derived from the captured real-user-behavior in software development (see Section 10.3).

We evaluated PlatinVC using a *testbed*. While the real prototype software was running on the machines in our lab, the user behavior was scripted with predefined events. This allowed us to repeat the same experiment setup multiple times for the sake of statistical correctness of the results.

The choice of this evaluation methodology showed its benefits very early. We discovered some design mistakes in the initial experiment that were not visible during our daily work with PlatinVC. Some of the behavior we initially assumed while designing the system, proved to be wrong in certain extreme conditions. That forced us to refine some mechanisms implemented in PlatinVC. With the other mentioned approaches, these incorrect assumptions might have never been discovered, as they would have been integrated in our models.

In the next subsections, the details of the evaluation environment and platform will be described.

10.2.1 *Evaluation Environment*

We carried out our experiments in two labs with connected computers and used 37 machines to run our experiments. Their hardware configuration was as following:

- first lab
  - 9 PCs with an Intel Core2Duo CPU clocked at 2.66 GHz, with 4 GB RAM
  - 12 PCs with an Intel Pentium P4 CPU clocked at 3 GHz, with 2 GB RAM
- second lab
  - 3 PCs with an Intel Pentium P4 CPU clocked at 3 GHz, with 3 GB RAM
  - 13 PCs with an Intel Pentium P4 CPU clocked at 2.8 GHz, with 2 GB RAM

The machines are connected to a switch in each lab, the two labs are connected with each other and the Internet via a switch. While this topology represents an ideal case, we were still able to see the trends of the prototype clearly.The operating system on all machines was Microsoft Windows XP with Service Pack 3. Additional software required was Java in version 1.6.0_20 and Mercurial in version 1.5.1.

In the experiments where we ran multiple instances of our prototype on a single machine, we deployed more instances on the stronger machines. Again, running multiple instances on one machine does not bring a realistic network topology, but shows a good approximation of the behavior in a real network. The uneven upload and download bandwidth of a peer in a wide area network (WAN) can be neglected, as the messages' payload was smaller than the minimum bandwidth available in this setup (assumed that the minimum bandwidth for uploading a message would be 16 kbps; the messages in our system have been maximal 15kbit big). The small message size is justified in the fact that only ***deltas*** to existing versions are transferred.

10.2.2 *Evaluation Platform*

In order to run controlled, repeatable accurate experiments, we developed an evaluation platform in [CG09]. It represents a distributed software that consists of a *controller* and a *client application*. After an instance of the client application is started on all machines, an instance of the controller application distributes to all machines the following:

- prototype of PlatinVC that is subject to evaluation,
- the data needed for the experiment, i.e., the module that stores the version history manipulated in the experiment,
- the users input into the prototype, i.e., actions it performs.

The users input was predefined in a xml file as detailed in Section 10.3.3. This evaluation platform allowed us to repeat an experiment under the same conditions, with the same user input, on easy and controllable fashion.

Each running instance of our prototype was measured by the accompanying client application of the evaluation platform. To avoid interference with the evaluation measurement, the distributed parts of the platform did not communicate during the experiment runtime. Recorded values, such as the result and runtime of executed operations, were stored in local log files which were collected after a specified experiment end-time.

We took into account unsynchronized local clocks and different network delays to be sure to start, execute, and end the experiment on any machine at the same time.

10.3 WORKLOAD

The workload we used in the experiments consists of the following aspects:

- size of the network,
- data used for versioning in the evaluated prototype, the module that stores the version history manipulated in the experiment,
- user behavior, and
- churn model.

In the following subsections we will discuss each of those aspects individually.

10.3.1 *Number of Users*

A survey [ISHGH07] shows that the number of users in the ***GSD*** scenario (see Section 2.3) is significantly smaller compared to other peer-to-peer application scenarios, like file sharing. The working force of a project was given in person/month. Most projects (33%) have a volume of under 10 person/month, the second 10-20 (22%) person/month. While there are projects with more than hundreds of person/month (16% 100-1,000 person/month, 3% more than 1,000 person/month) the average project size is 84 person/month. Two person/month could stand for one person, who worked two month long, or two persons, working for a month. A project's runtime was not provided, so we can only approximate the actual number of software developers involved. Assuming a 6 month project runtime, which is in most cases the minimal runtime of a project, we can conclude that, on average, 14 persons are simultaneously working on a ***GSD*** project.

Therefore, and with regard to the physical limitation of real machines, we set up our experiments with

- 3 (all peers are replicas of each other) users,
- 15 (typical project size in real GSD projects) users,
- 37 (limit of available machines for our experiments) users,
- and 100 (3 instances running on a single machine) users.

We did not scale our experiments to more than 100 users as the results would have been less accurate. The reason is that, with 100 users, we deployed three instances of our prototype at the faster machines and two at the slower computers. Deploying multiple instances on the same machine does cause moderate undesired side effects, such as delayed process execution, resulting from overloaded processors.

10.3.2 *Experiment Data*

We used the version history of the open source project gcc[1] as a realistic project module. Subversion is used in the development of gcc, however, there is a Mercurial mirror module accessible under `http://gcc.gnu.org/hg/gcc/trunk`, which we used.

The module's size is 654.7 MB large, it has 99,322 snapshots stored. The snapshots cover the project's history from the 23rd of November 1988, the project's start date, to the 12th of April 2010, two days before the release of version 4.5.0. A median of five files, each in a different folder, which belong to two branches in the directory tree, were changed in a snapshot. In our experiments every peer had an initial copy of this module, so that only updates made during the experiment were transferred.

1 http://gcc.gnu.org/

10.3.3 *User Behavior*

We derived the user actions from the log of the *gcc project*. We chose a busy timeframe of three hours to replay the ***commit*** commands which happened on Monday, the 12th of April 2010. The version 4.5.0. of gcc was released two days later. However, from the log we could only retrieve the commit operations that were executed using Subversion. For 50% of the commit operations of the log we executed operation *O-9: Push globally* in our experiments. The other 50% of the commit operations were first locally committed and pushed globally when a subsequent commit occurred.

The ***update*** operation is not recorded in a Subversion log, therefore we decided ourselves when to ***pull*** for the latest versions. In order to evaluate the ***freshness***, we executed many pull operations in a short time. Starting only four seconds after the initiation of a push command, every 0.5 seconds a (different) random user *pulls* the latest changes 20 times, which is equal to a period of ten seconds. The pulling users use operation *O-6: Pull globally* in one and operation *O-5: Pull globally using specified folders* in another setup. The concrete scenario is detailed with each result in the following subsection.

10.3.4 *Churn Model*

We used two models to emulate ***churn*** in our experiments. As detailed in Section 2.3, we assumed a mixture of open source developers and professional developers working on a ***GSD*** project.

Open source developers (as well as wiki users) act similarly to the captured behavior of file-sharing users. They tend to be online for a short amount of time. The longer they stay online, the less likely it is that they go offline. This behavior was analyzed in [SENB07a]. The online time of a user follows a Weibull distribution with the parameter values 357.7152 for *scale* and 0.54512 for *shape*. With theses parameters the Weibull distribution is similar to an exponential distribution, for which the tail of the curve decreases more slowly. We reused the implementation of this churn behavior provided by PeerfactSim.KOM [KKM+07].

To model the behavior of professional developers we introduced a second churn model. We assume that the arrival time of professional developers at their working place follows a normal distribution and that they stay online during their normal working hours. With the same normal distribution they switch off their machines and finish their working day. In a real company the developers are often directed not to switch off their machines.

Figure 59 visualizes how we simulated ***churn*** in our experiments. With churn 25% of the users leave and join the peer-to-peer network according to the Weibull distribution, which represents the open source developers (the topmost curve in the graphic). In the first hour only a few open source developers are online. Some are leaving the system and some are joining. The longer an open source developer stays online the less likely it is that he will leave, thus the number of developers online rises over time. The rest represents professional developers in different time zones, with different working hours. As our simulations ran for 3 hours and were not spanning an entire day, we decided that 25% of the developers stay online (not depicted in the graphic), while 25% leave the system and the remaining 25% join, to capture the effect of different time zones. The leaving developers start to leave after an hour runtime, and leave in the timespan of an hour. In the same timespan, the joining developers begin their work and are complete after another hour. They stay online until the end of the experiment afterwards, while the developers who left, stay offline. Only the open source developers, whose behavior is modeled using the Weibull distribution, leave and rejoin the network in an alternating fashion.

10.3.5 *Experiment Timeline*

Each experiment ran for 3 hours real time with a repeating sequence of push and pull commands. We carried out 20 repetitions of the same setup in order to meet statistical

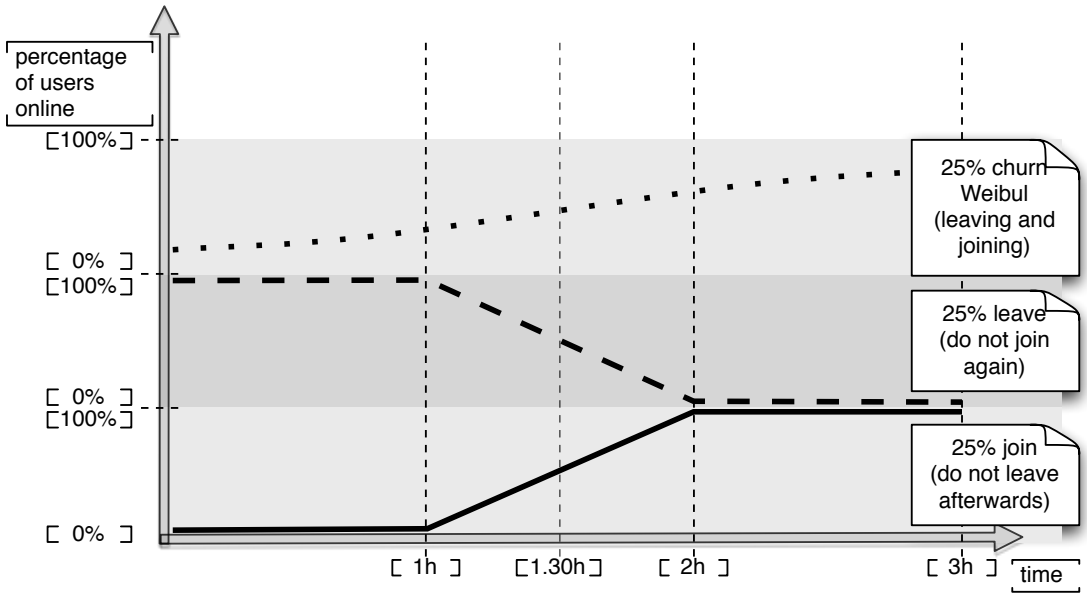


Figure 59: Churn execution in our experiments

correctness of the experiments. Within this 20 repetitions we varied the number of users only (3, 15, 37, 100). The measurements were executed after a start-up time, when all peers joined. We compared three different setups. A third of our experiments ran with no leaving peers, another third with the churn model described in Section 10.3.4, and the last third with failing peers. In the last setup 50% of the peers simultaneously left unannounced (failed) in the middle of the experiment.

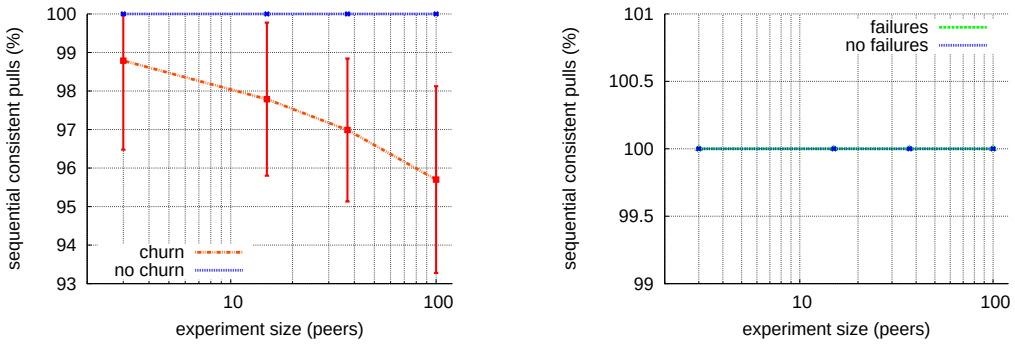
10.3.6 *System Parameter Settings*

We parameterized PlatinVC in all experiments with the following settings:

- The number of ***replica peers*** was set to 3. N.B. Most ***peer-to-peer applications*** use five peers. As detailed in Chapter 8 we store the data in a more redundant manner than, e.g., file sharing solutions. Thus it can be recovered more easily in the case that all replica peers fail in the same time window. However, to avoid this situation the number of replicating peers should be higher in a network with more ***churn***.
- The *time-outs* were set to 30 seconds for all ***pull*** and 60 seconds for all ***push*** operations. That matches 6 times the duration this operations needed to complete in preliminary tests.

10.4 EVALUATION RESULTS

The final results of our testbed-experiments are detailed in this section. Each subsection discusses one ***quality aspect***, with the exception of the aspects ***scalability*** and ***robustness***, which are presented in every experiment result. Both of these quality aspects are inherent aspects of a system, which are present all the time. To show the scalability of PlatinVC, we repeated each experiment setup with a different number of participating peers. To address robustness, we executed two thirds of the experiment duration using the churn model described in Section 10.3.4. The graphs present the mean value measured in each experiment, where the confidence intervals represent the variation of different experiments.



(a) with and without churn

(b) with and without simultaneous failure of 50% of the peers

Figure 60: Degree of consistency

10.4.1 *Consistency Degree*

Figure 60 shows that our assumption about the degree of consistency PlatinVC provides was pessimistic. When the network is stable ***sequential consistency*** is 100% guaranteed. No pull operation retrieved an ***outdated*** snapshot. N.B. Consistency is calculated only for the successful operations, as presented in Section 4.4. Figure 61 shows the relation of successful and unsuccessful operations, which did not retrieved any snapshots.

Even when half of the participating peers simultaneously fail, as presented in Figure 60b, PlatinVC maintains its ***sequential consistency***. Although a huge number of maintaining and replicating peers failed and with them some snapshots got lost, the peer who pushes a new snapshot stores all ***basis snapshots*** as well, and reintroduces them into the system. Only if a snapshot has been pushed, successfully pulled once, and got lost due to failing peers before it could be pulled a second time, the consistency degree would drop, which did not happen in any of our experiments. Due to the fact that all peers are failing at the same time either all snapshots or none are retrieved (which is the criteria for sequential consistency). By consulting Figure 61 we can see that no snapshots where retrieved in only 8% of the operations in the worst circumstances.

Introducing churn, as described in Section 10.3.4, decreases the number of sequentially consistent pulls to 93% in the worst case scenario. The remaining 7% showed to be still ***causally consistent***. In opposite to the previous scenario users are joining as well, and peers are failing at different times. Thus it is more likely that one snapshot can be retrieved, while the retrieval of a second snapshot in a branch fails, leading to ***causal consistency***. With a rising network size the percentage of sequential consistent pulls drops, because more peers act as maintainers and are involved in the version control operations. If all snapshots were concentrated on a single peer, either the peer is available and all snapshots can be retrieved, or none, thus complying to the definition of causal consistency. When the snapshots are distributed among multiple maintainer if only one if unavailable and all but one snapshot is retrieved only causal consistency is provided.

These results prove robustness and scalability of PlatinVC regarding its consistency.

10.4.2 *Robustness*

Figure 61 shows that a number of push and pull operations failed to complete in our experiments. For more then 15 peers less than 2% of the operations failed even without leaving or joining peers. A closer look on the evaluation results revealed that higher timeout values would

have prevented the operations from failing. In the experiments where half of the peers failed at once we experienced aborted operations in all setups, from 2.5% to 8%, in a logarithmic curve. Similarly to the degree of consistency the operations are less successful under churn

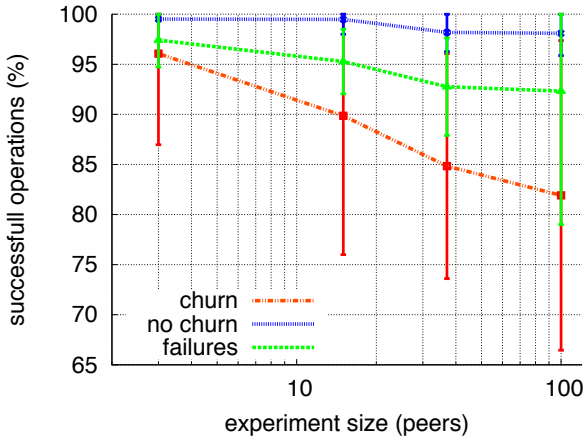


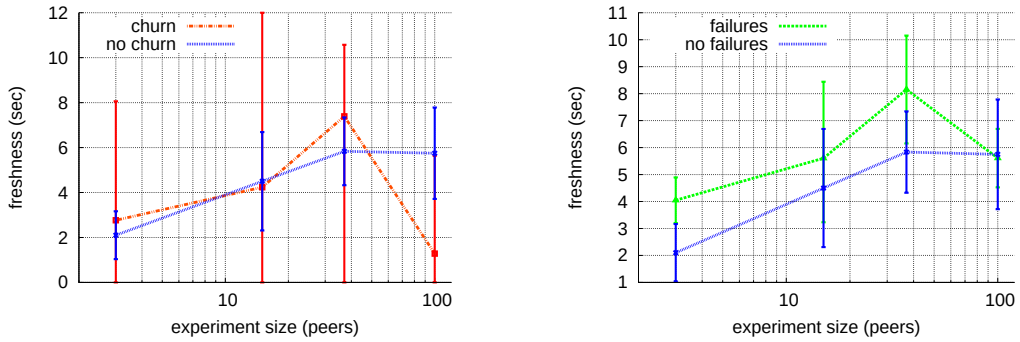
Figure 61: Percentage of successful operations

than when peers fail at once, as the returning peers in the churn scenario are contacted, but are not updated yet. Thus the operation fails. When peers fail without returning the remaining peers take over, already up to date due to the replica maintenance (see Section 8.4.6).

Again, with a growing network size more operations are aborted, because the repository is distributed among more peers which become unavailable. In our experiments for a network size up to 37 real machines in the worst case scenario only 25% of the operations failed and would have to be repeated. The slow grows of the percentage of unsuccessful operations proves the robustness of PlatinVC.

10.4.3 *Freshness*

Figure 62 shows that a snapshot is available, at the latest, 6 seconds after it was pushed. As the freshness value only rises slightly, we can conclude good scalability of PlatinVC regarding



(a) with and without churn

(b) with and without failing peers

Figure 62: Freshness of pushed updates (time till available)

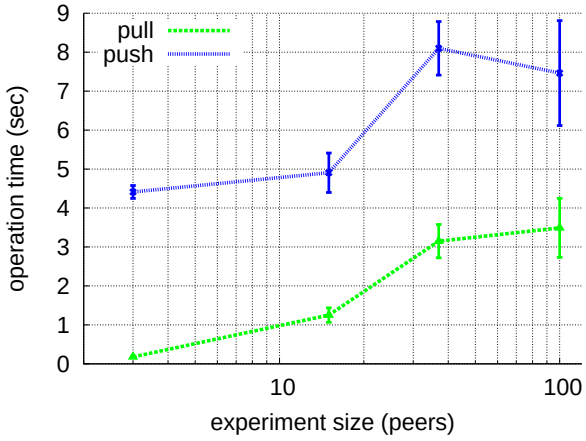


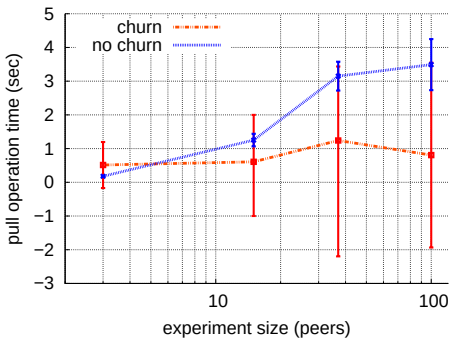
Figure 63: Time needed to execute the push/pull operation

its ***freshness***. The peer-to-peer system dynamics caused by churn or simultaneous failure of peers, proved to have only insignificant influence on the ***freshness*** of the retrieved updates. The values measured for 100 peers indicate that running multiple instances of our system on a single peer disturbs our measurement; whenever an instance running on the same machine pulls a snapshot pushed by an instance on the same machine the communication time is very fast, as messages are not sent over the network connection. In our experiment where all peers where stable (no churn or no failing peers) we can see that there is no significant difference between the results of the experiment with 37 peers or 100 peers, both running on 37 computers. A closer look on the experiment log files revealed that even in the experiment with 100 peers no to peers on the same machine communicated. In the experiments where we introduced churn and failing peers it happened that the replacing peers where instances on the same machine, which communicated with each other, leading to even faster updates.

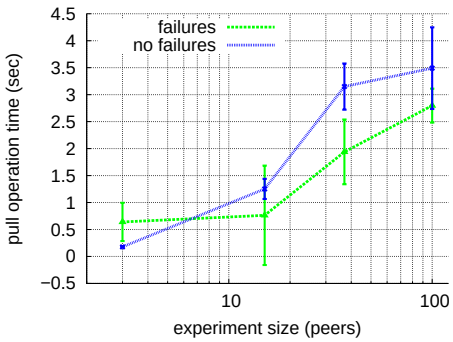
10.4.4 *Duration of Push and Pull*

As detailed in Section 8.4.4, there is always a trade-off when designing a version control system: Performance increase of one operation decreases the performance of the other, in the case of sharing or retrieving operations. Figure 63 clearly reflects our design decision to favor the performance of pull operation, as more frequently used operation. Executing the push operation takes between 4 and 5 seconds and can rise to up to 9 seconds, when the system grows. The very small amount of time needed for a pull operation of under 2 seconds, increasing with the network size to up to 4 seconds, is influenced by situations, in which the peer that is a ***replicating*** of ***maintaining peer*** already received the latest snapshots before it executed the pull operation. In our experiments a snapshot consisted of changed files being in five different folders. With two replicating peers and a maintainer for each folder in the most extreme case 15 peers would get the snapshot during the push operation.

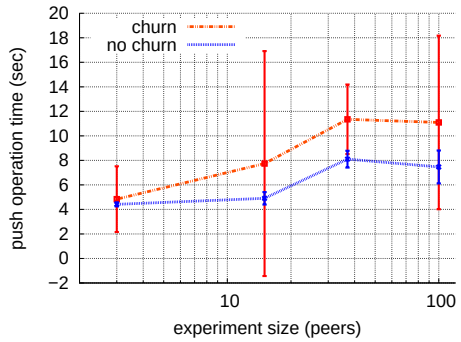
When the network grows to 100 peers, the operation time even drops. We do not consider this as the realistic behavior of our systems but rather a limitation of our experiment environment. We evaluated 100 peers by running multiple instances distributed among our 37 physical machines. Whenever two instances that run on the same physical machine communicate, the message exchange is unrealistically fast, which results in a shorter overall operation time. We can see the slight distortion of the evaluation results when emulating more peers than available physical machines in the evaluation environment.



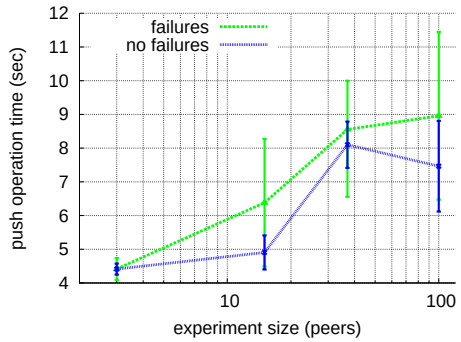
(a) Pull with and without churn



(b) Pull with and without simultaneous failure of 50% of the peers

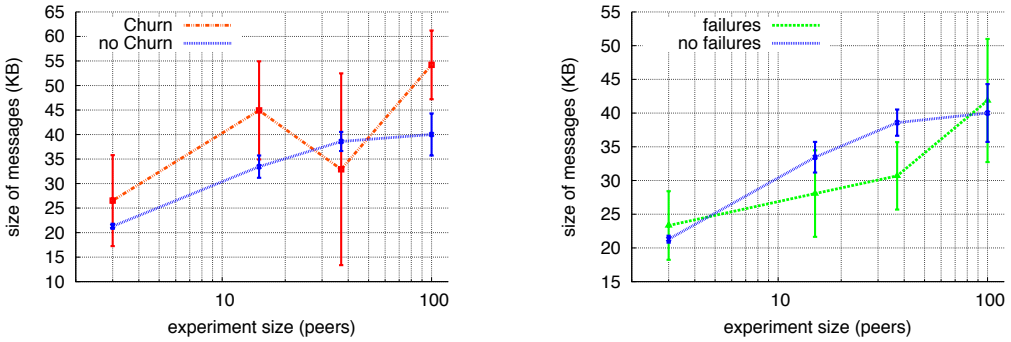


(c) Push with and without churn



(d) Push with and without simultaneous failure of 50% of the peers

Figure 64: Operation time



(a) with and without churn

(b) with and without failing peers

Figure 65: Size of transferred messages (payload+system messages)

Figure 63 shows the operation times when the participant number is not changing, when there is no churn. In Figure 64 we can see the operation performance under churn and when half of the peers simultaneously fail. The same trend being visible in the graphs, both with and without churn and failure shows that PlatinVC provides a good robustness. Scalability in the terms of operation time is good for both push and pull, even though it shows some rising trends in the case of peer failures.

The pull operation times tend to be better under churn and with failing peers. We can explain this behavior by the fact, that a smaller total number of peers is active in these scenarios. We did only measure pull requests which succeeded - as expected the number of successful requests drops, as shown in Figure 61. The very small values we measured for the operation time depends on the network structure as well. Whenever the requesting peer is a replicating peer of one of the artifacts under request, the items are available (due to the replication process) before the operation finishes. The smaller the network is, the more likely an arbitrary peer is a replicating peer of one of the artifacts in question.

10.4.5 *System Load*

Figure 65 further shows that with a rising number of participants, the summed up size of all messages is rising. This can be explained by the fact that with more peers in the system, the folders in a module are more widely distributed. More replicating peers have to be updated, which results in more messages. Additionally, the connectivity among the peers is smaller and more intermediate hops are needed to transfer a message.

A noticeable exception is the value for 37 peers under ***churn***. The summed message size drops below the value observed for 15 peers. However, the variance of the measurement is very large as well. This chaotic behavior is expected when a system experiences heavy churn.

With a growing number of peers the system load rises approximately logarithmic, indicating that the system load scales well.

10.5 COMPARATIVE EVALUATION

There is hardly any version control system, which exhibits the features PlatinVC demonstrated (see Chapter 6). The ***dVCS*** are lacking a ***global repository*** while a ***cVCS*** would collapse under churn, if the server machine fails. We were not able to run the only comparable peer-to-peer based solution Pastwatch (see Section 5.3.5) on more than 10 machines.

Nevertheless, we made some simple experiments to obtain comparative values for ***cVCS***'s: We measured the time needed to update the working copy using Subversion (see Section 5.1.4)

	SVN	Pastwatch	**PlatinVC**
commit / push	5.9 sec	3.74 sec	**4.8 sec**
update / pull	21.4 sec	2.7 sec	**1 sec**

Table 4: Comparison of commit/push and update/pull operation time in Subversion (SVN), Pastwatch [YCM06] and PlatinVC

with the same repository that we used for our measurements: gcc. It took 23 seconds (in mean) to update an ***outdated*** working copy to the latest version (updating revision 161333 to revision 161334). To not interfere with the development of this open source project, we measured the time needed to commit changes made in our Subversion repository: We controlled the versions of our evaluation platform in a Subversion repository hosted by google code, which utilizes Amazon's s3 servers. Obtaining the latest updates took, in mean, 21.4 seconds, while committing a 4.6 kb file took in mean 5.9 seconds, with a variation of 0.6 seconds in both operations.

As we could not run Pastwatch with more than ten peers, we could compare it to PlatinVC only by referring to the evaluation results from its conference publication [YCM06]. The commit operation needs 3.75 seconds in mean and the update operation 2.7 seconds in mean to perform. However, in the simulation setup only ten machines were used with two users, which is only comparable to our setting in which we used three peers with three users. In the experiments, where the number of project members was increased to 200, the number of machines did not change. These experiments showed a mean time needed for the commit operation of 5.45 seconds and 4.6 seconds for the update operation. However, we do not comprehend in which realistic scenario 200 users use only ten machines. Additionally, in those experiments only two users executed the operations. As pointed out in Section 5.3.5, we believe that it suffers from significant scalability issues when multiple users execute the offered operations in a small amount of time, as a centralized ***index*** has to be contacted each time.

As we can see in Table 4, in addition to the many benefits of PlatinVC discussed in Chapter 6, the performance of PlatinVC version control operations is also significantly better than in Subversion and is comparable to Pastwatch. An overall comparative evaluation of the offered features is summarized in Table 5.

10.6 SUMMARY

Our testbed evaluation showed the consistency degree of PlatinVC to be ***sequential*** in almost all cases. Even under ***churn***, 93% of the executed pull operations retrieve the latest changes, in the very worst case. The time needed to execute a push is fast with up to 5 seconds for 15 peers in system and 9 seconds for 37 peers. The pull operation is even faster with up to 2 seconds for 15 and 3 for 37 peers in the system. PlatinVC proved to be robust and scalable regarding consistency, freshness, and performance, whereas the scalability proved to be average in the case of the performance of pull operation.

Our experiments efficaciously modeled the global software development scenario (see Section 2.3), in which the average number of participating project members is 14. All results from the experiments with 15 peers prove PlatinVC's suitability for ***GSD***. In spite of the fact that our experiments did involve more than 100 peers, we cannot make a clear statement about the suitability of PlatinVC to the wiki scenario (see Section 2.2) with the thousands of users. The modifications of the articles in that scenario, however, is significantly less frequent than in our experiments and involves only a few users. With the scalability not evaluated to the maximum our experiments still showed that both application scenarios benefit from the other advantages the peer-to-peer paradigm can bring, as summarized in Section 2.4. Therefore, we

	cVCS	dVCS	Pastwatch	**PlatinVC**
Features		Offline version control	✓	✓
		Bisect debugging	✗	✓
	✗	Interoperability with other VCSs	✗	✓
		Tool support (i.e. history visualization)	✗	✓
	Selective updates	✗	✗	auto isolation selective updates
	✗	Personal commits	✗	combined
	Global commits	✗	✗	combined
	✗	✗	✗	Traceability links
Snapshots	most modern	✓	✓	✓
Consistency	***sequential consistency***	***eventual consistency***	***eventual consistency***	***causal consistency***

Table 5: Comparison of the features offered by the observed version control systems

can conclude that with our experiments, we placed PlatinVC under extreme workload for the observed scenarios and proved it to fulfill all requirements.

Part IV

FINALE

This part presents the final summaries and conclusions of our findings in the presented work and its contributions to and impact on future research in the area of collaborative and distributed project development.

11 CONCLUSION

This chapter summarizes and concludes the previous chapters (Section 11.1) and lists the main findings and contributions of this thesis (Section 11.2). In Section 11.3 we provide an outlook on open research challenges and in Section 11.4 we discuss the implications of the research presented in this thesis on software development environments and supporting applications.

11.1 SUMMARY AND CONCLUSIONS

Globally distributed software development is common today. Efficient and accurate collaboration between developers spread all around the globe is essential for the success of projects. In order to provide appropriate support for this, a version control system is a vital part of any collaboration platform. Its main functions are to track the evolutionary changes made to a project's files and manage work being concurrently undertaken on those files. In Chapter 1 we discussed the lack of current version control systems, their unsuitability to the global software development due to inefficient client-server communication, the presence of single point of failure, and scalability issues. The vision of this thesis is a decentralized, peer-to-peer version control system that overcomes the following challenges:

- *Lack of global knowledge* which is evidently crucial when pushing and pulling updates and finding the appropriate group of the peers for which the updates are relevant. Finding a particular peer in a fully decentralized fashion, with no centralized view on the system is already a challenging task.
- *Reliability* of the system and its performance due to the unpredictable peer behavior. How to create a reliable system on the back of unreliable, autonomous peers?

In Chapter 2 we discussed shortcomings of client-server based solutions for applications with users spread around the globe and the benefits that the peer-to-peer paradigm can bring. We analyzed two application scenarios in which version control is crucial: Wikis and global software development. A running example is described in which both application scenarios are integrated but the focus lies in global software development, as it is the more demanding application.

In chapter Chapter 3 we derived and analyzed a complete set of requirements for a peer-to-peer version control system that meets all the needs of collaborative development today. A list of assumptions, along with the exhaustive requirements listed in this chapter gave a clue as to the general applicability of the solution presented in this work.

As a first step in evaluating existing version control systems, in Chapter 4 we looked at the foundations of version control systems: workflows (***centralized collaboration workflow***, ***integration manager collaboration workflow*** or the ***lieutenants collaboration workflow***), the impact of the frequency of sharing changes, and configuration management. The special focus of this chapter was dedicated to specify and formally define key quality aspects of version control systems: the different levels of ***consistency*** and ***coherency***.

In Chapter 5 we explained in detail the most important version control systems, categorized according to the centralization of communication: ***centralized (cVCSs)***, ***distributed (dVCSs)*** and ***peer-to-peer based*** version control systems.

We discussed the limitations of those solutions, which include but are not limited to the following points:

- The centralized solutions bring unbearable delays for some developers as well as a single point of failure.

- The distributed systems are unable to share the produced snapshots effortlessly with all developers.
- The peer-to-peer based solutions tend to solve these issues but cannot provide a reliable degree of consistency.

In Chapter 6 we present the detailed comparative evaluation of existing version control systems. Our special attention is given to their design decisions and effects on key quality aspects - consistency and coherence.

Based on the finding from the previous chapters, our solution, peer-to-peer version control system PlatinVC is developed. First in Chapter 7 we give an overview of the architecture of PlatinVC, with the following major points:

- Not relying on a single participant, like usual in dVCSs, while offering the centralized up to date view of a cVCS.
- Collaboration workflows of cVCSs can be used in addition with local working practices of dVCSs, which results in the novel workflow presented in Section 7.2.
- PlatinVC is based on a dVCS which can be full utilized by the user. It inherit's the ability to work without network connection, automatic branches and interoperate with selected version control systems and introduces support for managing the evolution of traceability links.
- Basic configuration management is enabled by snapshot based version control and links between files, folder and modules.
- Changes in a working copy are stored as snapshots. They can be first collected locally to be later pushed globally. Once pushed the snapshots can be pulled by every participant immediately.
- Snapshots are pushed to a limited number of peers, where they are retrievable immediately. By pulling from those peers when needed all peers eventually store all snapshots locally, providing minimal access times.

In Chapter 8, we elaborate each design decision made and analyze all alternatives. Rather then reinventing existing solutions PlatinVC integrates software that proved to be stable in many other projects, and focuses on implementing the new parts. Main design decisions are:

- Our solution abstracts from a concrete ***peer-to-peer overlay protocol*** and only requires a solution that provides key based routing, which is implemented by any ***structured peer-to-peer protocol***.
- The modification on a user's working copy is tracked and stored using *Mercurial*, a decentralized version control system (see Section 5.2.4). As a side effect all tools that support or extend Mercurial, e.g., to visualize the recorded version ***history***, can be used with PlatinVC as well.
- In overcoming Mercurials limitations, which result in a lower ***degree of consistency***, we developed mechanism to share committed snapshots with all fellow developers. As a result PlatinVC combines the benefits a *locally stored version history* brings with the *automatic and complete collaboration* of centralized solutions.
- With the sophisticated distributed mechanisms presented in this chapter we were able to combine the best features of centralized and distributed version control systems while eliminating their drawbacks.
- We carefully considered situations which can lead to a faulty execution and designed mechanisms that counteract those effects and maintain a highly consistent and available ***repository***.

We developed a running prototype of PlatinVC and present it in Chapter 9. Peer-to-peer version control is only a part of the project support environment PIPE, we developed. It represents a modular framework that encapsulates services in components. The standardized inter-component communication is handled by the OSGi implementation *Equinox*. It also includes a component that handles synchronous and asynchronous communication and brings some awareness of the coworkers related contributions (ASKME). Another major component is the security manager, which handles secure message exchange and provides a role based access control mechanism.

In Chapter 10, we presented our testbed evaluation of PlatinVC. The evaluation results showed that:

- The consistency degree of PlatinVC is nearly always ***sequential***. Even under ***churn***, 93% of the executed pull operations retrieve the latest changes, in the very worst case.
- PlatinVC proved to be robust and scalable regarding consistency, freshness, and performance.
- The time needed to execute a push is minimal with up to 5 seconds for 15 peers in system and 9 seconds for 37 peers. The pull operation is even faster with up to 2 seconds for 15 and 3 for 37 peers in the system.
- In addition to the many benefits of PlatinVC discussed in Chapter 6, the performance of version control operations is also significantly better than in Subversion and is comparable with Pastwatch.

Our experiments effectively modeled the global software development scenario (see Section 2.3), in which the average number of participating project members is 14. All results from the experiments with 15 peers prove PlatinVC's suitability for ***GSD***. In spite of the fact that our experiments did involve more than 100 peers, we cannot make a clear statement about the suitability of PlatinVC to the wiki scenario (see Section 2.2) with the thousands of users. The modifications of the articles in that scenario are, however, significantly less frequent than in our experiments and involve only very few users. Therefore, we can conclude that with our experiments we placed PlatinVC under extreme workload for the observed scenarios and proved it to fulfill all requirements.

11.2 CONTRIBUTIONS

The main contributions of this thesis are (in the order of presentation):

- Taxonomy and exhaustive comparative evaluation of existing version control systems.
- Formal definition of consistency degrees of version control systems.
- A fully decentralized, peer-to-peer version control system that provides required consistency and coherence levels.
- Analysis of all options for design decisions in peer-to-peer version control systems.
- Solution to the "frequency-of-commit" problem by recording personal snapshots locally first and globally later, when changes are final.
- Automatic branches, which are created when needed, i.e., when conflicts arise.
- A running prototype of PlatinVC.
- The global software development environment PIPE.
- A modular framework for development of peer-to-peer based applications. PIPE provides abstractions and messaging modes which can be easily used without having a deep peer-to-peer knowledge.

- An evaluation methodology for version control systems consisting of an evaluation platform and detailed workload derived from the real user behavior and data.
- Proving that, in spite of unpredictable network dynamic and lack of centralized view, the peer-to-peer communication paradigm can achieve quality aspects that seemed to be possible only in centralized solutions, such as consistency.

11.3 OUTLOOK

PlatinVC proved to be an effective solution. In the later phases of its development we used it for our own version control, e.g., evolution of this thesis was tracked using PlatinVC. Based on the presented mechanisms, however, our version control system could be extended for usage in the following alternative ways:

Seamless Version Control: The three locations, where versions are stored on a local peer (***working copy***, ***local module*** and ***peer-to-peer module***) could be exploited to make a version control as automatic as possible. Modifications to the artifacts in the working copy could be ***pushed*** automatically to other peers, where they would be applied to the peer-to-peer module. After a notification, a user could decide to pull those changes to his local module (executed in milliseconds as all updates are already locally present) and update his working copy. This proactive distribution of the updates could be implemented in a controlled way, as the only unpredictable factor would be when updates are created, but not if a user requests them. Rather than ***flooding*** updates as soon as they are created, sophisticated content distribution techniques could be utilized as proposed by [CDKR02]. Using the approach presented in [GSR+09, Gra10] the system could monitor the traffic and decide when to distribute updates to which parts of the network.

However, doing so would change the ***maintainer based repository distribution*** to the ***replicated repository distribution*** and worsen the ***consistency degree*** as a consequence to provide only ***eventual consistency***. Therefore, instead of relying on proactive updates, the proposed maintainer-based mechanisms could be extended with additional content distribution techniques that broadcast updates to all peers. A user would still be required to check for updates manually, but the updates would have most probably already been transferred proactively to a user's machine and the process would finish faster. However, the improvement might be insignificant, as our evaluation showed that updates do not lead to a large message overhead which takes a long time to be transferred.

Two Tier Setup: PIPE (see Chapter 9) could be extended to communicate with *client* machines, which do not offer services or resources but only consume them. The peers in the network would operate as servers for them, providing updates on demand and forwarding modified files. This setup could be helpful in the wiki scenario (see Section 2.2), in which some participants may not want to modify the articles but only consume some of them. The fact that all client machines would operate under the same ***peer identifier***, remains subject to further investigations.

Usage Based Replication: Even when projects are developed in collaboration by globally distributed developers some parts might be exclusively modified by the same group of developers. PlatinVC could, therefore, benefit from a location aware peer-to-peer overlay (like Globase.KOM [KHLS08, KLS07, KTL+09, Kov09]). The geographical location information would allow artifacts to be maintained by peers which are physically close to the machines which update them the most (or even exclusively). The more peers there are in one location updating a specific artifact, the more likely it would be maintained by one of them. However, these changes would require an ***indexing*** mechanism, as artifacts could not be found using locally available information (i.e. by calculating the ***hash value*** of an artifact's name). Additionally, dynamic changes that involve a large amount of data to be copied over to a different peer could worsen the overall performance of the system and neglect the possible gains.

Going beyond geographical optimization, we can organize storage and replication of artifacts also according to their current and predicted usage. For example, access statistics for artifacts can be protocoled and embedded in metadata. Then those metadata could be use to define some virtual artifacts landscape, using a graph embedding algorithm such as the spring embedder [Ead84] with force-directed placement. A graph is drawn iteratively until an equilibrium of forces between graph nodes (representing the location of artifacts in the landscape), is reached. Attracting and repelling forces are derived from metadata, which integrate compressed protocoled usage statistics.

Incorporate an Ad-Hoc Grid: To improve the stability and increase the system's performance grid machines could be integrated in the peer-to-peer network like described in [SFF04]. While the companies in our GSD scenario could provide computers, which only run PlatinVC in order to improve the system's performance, deploying a grid system on them would allow a better control. In this way a minimum service quality could be guaranteed, which is not possible when the system is running purely on a peer-to-peer network. While the solution in [SFF04] includes a peer-to-peer protocol implementation both, the proposed solution and PlatinVC would have to be modified. The peer-to-peer implementation used in the ad-hoc grid system is JXTA, which is not compliant to the key based routing API used by PlatinVC. It would be better to replace it with FreePastry, which is the peer-to-peer protocol used in our solution. PlatinVC would have to be adopted to the ad-hoc grid, so that the grid machines could be fully exploited. These machines could take over parts of any functionality; from storing a partial backup of the repository to acting as a maintainer for specified modules.

11.4 IMPLICATIONS

We saw in the example of a Wiki engine (described in Section 2.2) that a version control system is a core component of many other applications. The benefits PlatinVC brings to software development could be introduced to a number of other applications.

Versioned File System: Recent file systems incorporate version control mechanisms to provide a user with an automatic lightweight backup solution. The most mature approach is ZFS which was introduced by Sun Microsystems in 2005 [SM06]. The mechanisms of peer-to-peer based version control systems like [KBC+00] and Ivy [MMGC02] could be combined with the mechanisms elaborated in this work to create a peer-to-peer based versioned file system. However, the research questions stated above under *seamless version control* would arise in this approach as well.

Configuration Management: The obvious next step in the evolution of PlatinVC would be an extension with mechanism for configuration management (see Section 4.3). Most of the required mechanisms would not need to communicate in a distributed fashion. With the ***metadata*** and traceability links (see Section 7.3.7), PlatinVC offers the possibility for configuration information to be stored and retrieved in a decentralized fashion. With an extensional configuration management approach, the traceability links would point explicitly to the artifacts belonging to a configuration, while the *intensional configuration management* approach presented by [Cag93] could be enabled using attributes which can be stored in an artifact's metadata.

Enable Traceability: Traceability links can be used for various additional features in a project's development. For example, let us observe artifacts from different development phases, such as requirement, design, implementation and testing. Using traceability links, we can trace back a test case from the last phase of development to the related artifacts from the earlier phases, such as code or requirements. This feature could be helpful in deciding which tests are to be executed if budget or time constraints limit the executable tests.

PlatinVC enables traceability links between artifacts in different modules. It handles and maintains updates of traceability links and their version history. Updates are handled and a version history for each traceability link is maintained. However, there is currently no support for creating links only: not automatically, assisted or manually. Links can currently only be create via PIKI, which will be fully supported and tracked as traceability links by PlatinVC. An approach as described in [Köno8] could be used to create or update links between artifacts (semi-)automatically, which could be stored in a decentralized fashion using PlatinVC. We thereby avoid a communication bottleneck in the case of multiple developers accessing links which point to an disjunct set of artifacts. Using a centralized database to store the links, like proposed by the mentioned work, would limit the concurrent access any link.

Change Propagation: A version control system manages the conflicts arising when multiple developers simultaneously edit the same artifact. The consequences of concurrent editing can also have an effect on other artifacts which is not detected by version control today. This can happen when, for example, a function that is defined in one source code artifact is used in another, and both artifacts are changed concurrently. PlatinVC could be enhanced with mechanisms that detect those conflicts occurring between two or more artifacts and warn users about it, assist them in the repairs, or automatically solving those conflicts. The necessary steps could be configured using an artifact's ***metadata*** and the provided traceability links, as described above (*Configuration Management*).

However, noticing that the occurrence of such conflicts could be delayed, as maintainers would have to communicate changes, developers could make changes offline and the network could be partitioned temporary.

Storage Based Application: Besides version control, PlatinVC provides a purely decentralized and reliable data storage, manages concurrent updates and handles churn and network instability. Many collaborative applications could use this reliable storage and benefit from its decentralized architecture. We elaborated an example, a Wiki engine, in Section 2.2, but also similar Internet based applications, such as forum software, e-mail services or news pages could exploit the mechanisms offered by PlatinVC.

[AAG+05] Karl Aberer, Luc Onana Alima, Ali Ghodsi, Sarunas Girdzijauskas, Seif Haridi, and Manfred Hauswirth. The Essence of Peer-to-Peer: A Reference Architecture for Overlay Networks. In *Proceedings of the International Conference on Peer-to-Peer Computing (P2P)*, pages 11–20, 2005.

[AH02] Karl Aberer and Manfred Hauswirth. An Overview on Peer-to-Peer Information Systems. In *Proceedings of the Workshop on Distributed Data and Structures (WDAS)*, 2002.

[All07] O.S.G. Alliance. OSGi Service Platform, Core Specification, Release 4, Version 4.1. *OSGi Specification*, 2007.

[ATS04] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A Survey of Peer-to-peer Content Distribution Technologies. *ACM Computing Surveys*, 36(4):335–371, 2004.

[Blo04] Rebecca Blood. How Blogging Software Reshapes the Online Community. *Communications of the ACM*, 47(12):53–55, 2004.

[BLS06] Elizabeth Borowsky, Andrew Logan, and Robert Signorile. Leveraging the Client-Server Model in Peer-to-Peer: Managing Concurrent File Updates in a Peer-to-Peer System. In *Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT-ICIW)*, pages 114–119, 2006.

[BM05] David E. Bellagio and Tom Milligan. *Software Configuration Management Strategies and IBM® Rational® Clearcase®: A Practical Introduction*. IBM Press, 2005.

[BRB+09] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. The Promises and Perils of Mining Git. In *Proceedings of the International Working Conference on Mining Software Repositories (MSR)*, pages 1–10. Citeseer, 2009.

[Bur95] James H. Burrows. *Secure Hash Standard*. National Institute of Standards and Technology, 1995.

[Cag93] Martin Cagan. Untangling Configuration Management: Mechanism and Methodology in CM Systems. In *Proceedings of the International Workshop on Configuration Management (SCM)*, pages 35–53, 1993.

[CCR04] Miguel Castro, Manuel Costa, and Antony I. T. Rowstron. Performance and Dependability of Structured Peer-to-Peer Overlays. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 9–18, 2004.

[CDK05] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley Longman, 2005.

[CDKR02] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony I. T. Rowstron. SCRIBE: A Large-Scale and Decentralized Application-Level Multicast Infrastructure. *IEEE Journal on Selected Areas in communications*, 20(8):1489–1499, 2002.

[CG09] Damian A. Czarny and Alexander Gebhardt. Entwicklung einer automatisierten Evaluationsplattform für verteilte Anwendungen. Bachelor's thesis, Real-Time Systems Lab, TU Darmstadt, 2009. Supervisor: Patrick Mukherjee.

[Che04] Benjie Chen. *A Serverless, Wide-Area Version Control System*. PhD thesis, 2004. Supervisor: Robert T. Morris.

[CHSJ03] Li-Te Cheng, Susanne Hupfer, Ross Steven, and Patterson John. Jazzing up Eclipse with Collaborative Tools. In *Proceedings of the OOPSLA Workshop on Eclipse Technology eXchange*, pages 45–49, 2003.

[Coh03] Bram Cohen. Incentives Build Robustness in BitTorrent. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems (P2PECON)*, May 2003.

[Col] CollabNet - The Leader in Agile Application Lifecycle Management. http://www.collab.net/.

[CSWH00] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability (DIAU)*, pages 46–66, 2000.

[CVS] CVS - Concurrent Versions System. http://www.nongnu.org/cvs/.

[Den84] Dorothy E. Denning. Digital Signatures with RSA and other Public-Key Cryptosystems. *Communications of the ACM*, 27(4):392, 1984.

[DH79] Whitfield Diffie and Martin E. Hellman. Privacy and Authentication: An Introduction to Cryptography. *Proceedings of the IEEE*, 67(3):397–427, 1979.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the ACM Symposium on Operating Systems Principle (SOSP)*, pages 205–220, 2007.

[dro] Dropbox - Online Backup, File Sync and Sharing made Easy. https://www.dropbox.com.

[Dui07] Mike Duigou. JXTA 2.0 Protocols Specification. https://jxta-spec.dev.java.net/JXTAProtocols.pdf, January 2007. Version 2.5.2.

[Ead84] Peter Eades. A Heuristic for Graph Drawing. *Congressus numerantium*, 42(149160):194–202, 1984.

[EC95] Jacky Estublier and Rubby Casallas. The Adele Configuration Manager. *Configuration management*, 2:99–133, 1995.

[Eck09] Claudia Eckert. *IT-Sicherheit: Konzepte-Verfahren-Protokolle (6th Edition)*. Oldenbourg Wissenschaftsverlag, 2009.

[equ] Equinox. http://www.eclipse.org/equinox/.

[fac] Facebook. http://www.facebook.com.

[FFRS02] Thomas Friese, Bernd Freisleben, Steffen Rusitschka, and Alan Southall. A Framework for Resource Management in Peer-to-Peer Networks. In *Proceedings of the International Conference NetObjectDays (NODe)*, pages 4–21, 2002.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[Fou] The Eclipse Foundation. Eclipse project. http://www.eclipse.org/eclipse/.

[FP] Freepastry. http://www.freepastry.org/.

[GHHR07b] Michael Geisser, Armin Heinzl, Tobias Hildenbrand, and Franz Rothlauf. Verteiltes, internetbasiertes Requirements-Engineering. *Wirtschaftsinformatik*, 49(3):199–207, 2007.

[GHHS07b] Michael Geisser, Hans-Joerg Happel, Tobias Hildenbrand, and Stefan Seedorf. Einsatzpotenziale von Wikis in der Softwareentwicklung am Beispiel von Requirements Engineering und Traceability Management. *Social Software in der Wertschöpfung*, March 2007.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[Git] Gitbenchmarks. https://git.wiki.kernel.org/index.php?title=GitBenchmarks%5C&oldid=8548.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *ACM SIGACT News*, 33(2):51–59, 2002.

[GMM+09] Kalman Graffi, Patrick Mukherjee, Burkhard Menges, Daniel Hartung, Aleksandra Kovacevic, and Ralf Steinmetz. Practical Security in Peer-to-Peer based Social Networks. In *Proceedings of the Annual Conference on Local Computer Networks (LCN)*, pages 269–272, 2009.

[GNUa] Diffutils - GNU Project - Free Software Foundation. http://www.gnu.org/software/diffutils/.

[gooa] Google Documents. http://docs.google.com.

[goob] Google Offices. http://www.google.com/intl/en/corporate/address.html.

[Gra10] Kalman Graffi. *Monitoring and Management of Peer-to-Peer Systems*. PhD thesis, Multimedia Communications Lab (KOM), TU Darmstadt, Germany, 2010. Supervisor: Ralf Steinmetz.

[Gro] Groove virtual office. http://www.groove.net/index.cfm/pagename/VirtualOffice/?home=hp-overview.

[GSR+09] Kalman Graffi, Dominik Stingl, Julius Rueckert, Aleksandra Kovacevic, and Ralf Steinmetz. Monitoring and Management of Structured Peer-to-Peer Systems. In *Proceeding of the International Conference on Peer-to-Peer Computing (P2P)*, pages 311–320, 2009.

[HA90] Phillip W. Hutto and Mustaque Ahamad. Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 302–311, 1990.

[Har01] Ann Harrison. The Promise and Peril of Peer-to-Peer. http://www.networkworld.com/buzz2001/p2p/, September 2001.

[HBMS04] Oliver Heckmann, Axel Bock, Andreas Mauthe, and Ralf Steinmetz. The eDonkey File-Sharing Network. In *Proceedings of the Workshop on Algorithms and Protocols for Efficient Peer-to-Peer Applications (PEPPA)*, pages 224–228, September 2004.

[HCRP04] Susanne Hupfer, Li-Te Cheng, Steven Ross, and John Patterson. Introducing Collaboration into an Application Development Environment. In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW)*, pages 21–24, 2004.

[HM02] Harald Holz and Frank Maurer. Knowledge Management Support for Distributed Agile Software Processes. In *Proceedings of International Workshop of Advances in Learning Software Organizations (LSO)*, pages 60–80, 2002.

[HM03] James D. Herbsleb and Audris Mockus. An Empirical Study of Speed and Communication in Globally Distributed Software Development. *IEEE Transactions on Software Engineering*, 29:481–494, June 2003.

[HP98] John Hennessy and David Patterson. *Computer Architecture. A Quantitative Approach (4th Edition).* Morgan Kaufmann Publishers, 1998.

[HPB05] James D. Herbsleb, Daniel J. Paulish, and Matthew Bass. Global Software Development at Siemens: Experience from Nine Projects. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 524–533, 2005.

[HR83] Theo Härder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computer Survey*, 15(4):287–317, 1983.

[HRH07] Tobias Hildenbrand, Franz Rothlauf, and Armin Heinzl. Ansätze zur kollaborativen Softwareerstellung. *Wirtschaftsinformatik*, 49(Special Issue):72–80, February 2007.

[HT] Junio Hamano and Linus Torvalds. Git - Fast Version Control System. http://git-scm.com/.

[HT07] Octavio Herrera and Znati Taieb. Modeling Churn in Peer-to-Peer Networks. In *Proceedings of the Annual Simulation Symposium (ANSS)*, pages 33–40, 2007.

[IBM] IBM Rational Software. http://www-306.ibm.com/software/rational/.

[ISHGH07] Timea Illes-Seifert, Andrea Herrmann, Michael Geisser, and Tobias Hildenbrand. The Challenges of Distributed Software Engineering and Requirements Engineering: Results of an Online Survey. In *Proceedings of the Global Requirements Engineering Workshop (GREW)*, pages 55–66, 2007.

[Jav] For java developers. http://www.oracle.com/technetwork/java/index.html.

[Jaz10] Jazz community site. http://jazz.net/, 2010.

[JG03] E. James Whitehead Jr. and Dorrit Gordon. Uniform Comparison of Configuration Management Data Models. In *Proceedings of the Software Configuration Management Workshop (SCM) (adjacent to the ICSE)*, pages 70–85, 2003.

[Joo] Joost - The New Way of Watching TV. http://www.joost.com/.

[JXY06] Yi Jiang, Guangtao Xue, and Jinyuan You. Distributed Hash Table Based Peer-to-Peer Version Control System for Collaboration. In *Proceedings of the International Conference on Computer Supported Cooperative Work in Design III (CSCWD)*, pages 489–498, 2006.

[KB04] Yoram Kulbak and Danny Bickson. The eMule Protocol Specification. Technical report, School of Computer Science and Engineering, the Hebrew University of Jerusalem, 2004.

[KBC+00] John Kubiatowicz, David Bindel, Yan Chen, Steven E. Czerwinski, Patrick R. Eaton, Dennis Geels, Ramakrishna Gummadi, Sean C. Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Y. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 190–201, 2000.

[KHLS08] Aleksandra Kovacevic, Oliver Heckmann, Nicolas Liebau, and Ralf Steinmetz. Location Awareness - Improving Distributed Multimedia Communication. *Special Issue of the Proceedings of IEEE on Advances in Distributed Multimedia Communications*, 96(1), January 2008.

[KKM+07] Aleksandra Kovacevic, Sebastian Kaune, Patrick Mukherjee, Nicolas Liebau, and Ralf Steinmetz. Benchmarking Platform for Peer-to-Peer Systems. *it - Information Technology (Methods and Applications of Informatics and Information Technology)*, 49(5):312–319, September 2007.

[KLS07] Aleksandra Kovacevic, Nicolas Liebau, and Ralf Steinmetz. Globase.KOM - A Peer-to-Peer Overlay for Fully Retrievable Location-based Search. In *Proceedings of the International Conference on Peer-to-Peer Computing (P2P)*, pages 87–96, September 2007.

[Kön08] Alexander Königs. *Model Integration and Transformation – A Triple Graph Grammar-based QVT Implementation*. PhD thesis, Real-Time Systems Lab (ES), TU Darmstadt, Germany, 2008. Supervisor: Andy Schürr.

[Kov09] Aleksandra Kovacevic. *Peer-to-Peer Location-based Search: Engineering a Novel Peer-to-Peer Overlay Network*. PhD thesis, Multimedia Communications Lab (KOM), TU Darmstadt, Germany, 2009. Supervisor: Ralf Steinmetz.

[KR81] H. T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[Kru03] Philippe Kruchten. *The Rational Unified Process: An Introduction (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[KTL+09] Aleksandra Kovacevic, Aleksandar Todorov, Nicolas Liebau, Dirk Bradler, and Ralf Steinmetz. Demonstration of a Peer-to-Peer Approach for Spatial Queries. In *Proceedings of Database Systems for Advanced Applications (DASFAA)*, pages 776–779, April 2009.

[Lam79] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[Lam98] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[Lam01] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[LC01] Bo Leuf and Ward Cunningham. *The Wiki Way: Quick Collaboration on the Web*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[Leb95] David B. Leblang. The CM Challenge: Configuration Management that Works. pages 1–37, 1995.

[LLST05] Xin Lin, Shanping Li, Wei Shi, and Jie Teng. A Scalable Version Control Layer in Peer-to-Peer File System. In *Proceedings of the International Conference on Grid and Cooperative Computing (GCC)*, pages 996–1001, 2005.

[Loe09b] Jon Loeliger. *Version Control with Git*. O'Reilly Media, 2009.

[Mac] Matt Mackall. Mercurial, a Distributed SCM. http://selenic.com/mercurial/.

[Mac06] Matt Mackall. Towards a Better SCM: Revlog and Mercurial. In *Proceedings of the Linux Symposium*, pages 83–90, 2006.

[MH01] Audris Mockus and James D. Herbsleb. Challenges of Global Software Development. In *Proceedings of the IEEE International Software Metrics Symposium (METRICS)*, pages 182–184, 2001.

[MH07] Monika Moser and Seif Haridi. Atomic Commitment in Transactional DHTs. In *Proceedings of the CoreGRID Symposium (CoreGRID)*, pages 151–161, 2007.

[MHK05] Eve MacGregor, Yvonne Hsieh, and Philippe Kruchten. Cultural Patterns in Software Process Mishaps: Incidents in Global Projects. In *Proceedings of the Workshop on Human and Social Factors of Software Engineering (HSSE)*, pages 1–5, 2005.

[Mil] Phil Milford. Suit challenges Microsoft's deal for Groove. http://seattlepi.nwsource.com/business/218502_msftgroove02.html.

[Mil97] Bartosz Milewski. Distributed Source Control System. In *Proceedings of the System Configuration Management, ICSE SCM-7 Workshop (SCM)*, pages 98–107, 1997.

[MKS08] Patrick Mukherjee, Aleksandra Kovacevic, and Andy Schürr. Analysis of the Benefits the Peer-to-Peer Paradigm brings to Distributed Agile Software Development. In *Proceedings of the Software Engineering Conference (SE)*, pages 72—77, Februar 2008.

[ML83] C. Mohan and Bruce G. Lindsay. Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions. In *Proceedings of the Annual ACM SIGACT-SIGOPS Symposium on Princiles of Distributed Computing (PODC)*, pages 76–88, 1983.

[ML07] Joseph Morris and Chris Lüer. DistriWiki: A Distributed Peer-to-Peer Wiki. In *Proceedings of the International Symposium on Wikis (WikiSym)*, pages 69–74, 2007.

[MLS08] Patrick Mukherjee, Christof Leng, and Andy Schürr. Piki - A Peer-to-Peer based Wiki Engine. In *Proceedings of the International Conference on Peer-to-Peer Computing (P2P)*, pages 185–186, September 2008.

[MM02] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 53–65, 2002.

[MMGC02] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A Read/Write Peer-to-Peer File System. In *Proceedings of the Symposium on Operating System Design and Implementation (OSDI)*, pages 31–44, 2002.

[mon] monotone. http://monotone.ca/.

[Mos93] David Mosberger. Memory Consistency Models. *ACM SIGOPS Operating Systems Review*, 27(1):18–26, 1993.

[MR98] Dahlia Malkhi and Michael Reiter. Byzantine Quorum Systems. *Distributed Computing*, 11:203–213, 1998.

[O'So9] Bryan O'Sullivan. *Mercurial: The Definitive Guide*. O'Reilly & Associates Inc, 2009.

[OUMI06] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data Consistency for Peer-to-Peer Collaborative Editing. In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW)*, pages 259–268, 2006.

[PAE03] Rafael Prikladnicki, Jorge Luis Nicolas Audy, and Roberto Evaristo. Global Software Development in Practice Lessons Learned. *Software Process: Improvement and Practice*, 8(4):267–281, October 2003.

[PAP06] Leonardo Pilatti, Jorge Luis Nicolas Audy, and Rafael Prikladnicki. Software Configuration Management over a Global Software Development Environment: Lessons Learned from a Case Study. In *Proceedings of the International Workshop on Global Software Development for the Practitioner (GSD)*, pages 45–50, 2006.

[Par72] David Lorge Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[PH03] Joon S. Park and Junseok Hwang. Role-Based Access Control for Collaborative Enterprise in Peer-to-Peer Computing Environments. In *Proceedings of the Symposium on Access Control Models and Technologies (SACMAT)*, pages 93–99, 2003.

[Pil04] Michael Pilato. *Version Control With Subversion*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004.

[Que09] Marcel Queisser. Zugangs- und Zugriffskontrolle für Peer-to-Peer Netzwerke. Master's thesis, TU Darmstadt, March 2009. Supervisor: Patrick Mukherjee.

[RD01b] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the International Conference on Distributed Systems Platforms (Middleware)*, pages 329–351, 2001.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A Scalable Content-Addressable Network. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 161–172, 2001.

[RGRK04] Sean C. Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*, pages 127–140, 2004.

[Roc75] Marc J. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering*, 1(4):364–370, 1975.

[RR06] Suzanne Robertson and James Robertson. *Mastering the Requirements Process*. Addison-Wesley Professional, 2006.

[SE05] Ralf Steinmetz and Klaus Wehrle (Edits.). *Peer-to-Peer Systems and Applications*. Springer, September 2005.

[SENB07a] Moritz Steiner, Taoufik En-Najjary, and Ernst W. Biersack. Analyzing Peer Behavior in KAD. Technical report, Institut Eurecom, France, 2007.

[SFF04] Matthew Smith, Thomas Friese, and Bernd Freisleben. Towards a Service-Oriented Ad Hoc Grid. In *Proceedings of the International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 201–208, 2004.

[Shio1] Clay Shirky. *Peer to Peer: Harnessing the Power of Disruptive Technologies*, chapter Listening to Napster, pages 19–28. O'Reilly & Associates, 2001.

[Skyb] Official Website of Skype. http://www.skype.com.

[SMo6] Inc. Sun Microsystems. ZFS On-Disk Specification. http://opensolaris.org/os/community/zfs/docs/ondiskformat0822.pdf, 2006.

[Šmio6] Darja Šmite. Requirements Management in Distributed Projects. *Journal of Universal Knowledge Management*, 1(2):69–76, 2006.

[SMK+01b] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 149–160, 2001.

[SMLN+03] Ion Stoica, Robert Morris, David Liben-Nowell, David Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *IEEE Transactions on Networking*, 11(1):17–32, 2003.

[Som10] Ian Sommerville. *Software Engineering (9th Edition)*. Addison-Wesley, 9. edition, 2010.

[Sop] SopCast - Free Peer-to-Peer internet TV | live football, NBA, cricket. http://www.sopcast.org/.

[Sou01] Cleidson R. B. De Souza. Global software development: Challenges and perspectives, 2001. http://citeseer.ist.psu.edu/457465.html.

[SRo6] Daniel Stutzbach and Reza Rejaie. Understanding Churn in Peer-to-Peer Networks. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement (IMC)*, pages 189–202, 2006.

[SSo5] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Survey*, 37(1):42–81, 2005.

[SSFo8] Christian Schridde, Matthew Smith, and Bernd Freisleben. An Identity-Based Key Agreement Protocol for the Network Layer. In *Proceedings of the International Conference on Security and Cryptography for Networks (SCN)*, pages 409–422, 2008.

[SSRo8] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris: Reliable Transactional Peer-to-Peer Key/Value Store. In *Proceedings of the ACM SIGPLAN Workshop on Erlang*, pages 41–48, 2008.

[SZRCo6] Ravi Sandhu, Xinwen Zhang, Kumar Ranganathan, and Michael J. Covington. Client-side Access Control Enforcement Using Trusted Computing and PEI Models. *Journal of High Speed Networks*, 15:229–245, August 2006.

[Tic82] Walter F. Tichy. Design, Implementation, and Evaluation of a Revision Control System. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 58–67, 1982.

[TKLBo7] Wesley W. Terpstra, Jussi Kangasharju, Christof Leng, and Alejandro P. Buchmann. Bubblestorm: Resilient, Probabilistic, and Exhaustive Peer-to-Peer Search. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 49–60, 2007.

[TM04a] Katsuhiro Takata and Jianhua Ma. GRAM - A Peer-to-Peer System of Group Revision Assistance Management. In *Proceedings of the International Conference on Advanced Information Networking and Applications (AINA)*, pages 587–592, March 2004.

[TM05] Katsuhiro Takata and Jianhua Ma. A decentralized Peer-to-Peer Revision Management System using a Proactive Mechanism. *International Journal of High Performance Computing and Networking*, 3(5):336–345, 2005.

[Tor] Linus Torvalds. Clarification on GIT, Central Repositories and Commit Access Lists. http://lists.kde.org/?l=kde-core-devel&m=118764715705846.

[TS06] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice Hall, 2006.

[TTP+95] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 172–182, 1995.

[UML09] OMG Unified Modeling Language (OMG UML), Superstructure. http://www.omg.org/spec/UML/2.2/Superstructure, February 2009.

[Voc09] Robert Vock. Entwicklung eines Transaktionsprotokolls für Peer-to-Peer basiertes Versionsmanagement. Master's thesis, TU Darmstadt, December 2009. Supervisor: Patrick Mukherjee.

[Wal02] Dan S. Wallach. A Survey of Peer-to-Peer Security Issues. In *Proceedings of the International Symposium on Software Security (ISSS)*, pages 42–57, 2002.

[WB96] Mark Weiser and John S. Brown. The Coming Age of Calm Technology. Technical report, 1996.

[wikb] Wikipedia, the free Encyclopedia that Anyone can Edit. http://www.wikipedia.org/.

[WN94] Robert Watkins and Mark Neal. Why and How of Requirements Tracing. *IEEE Software*, 11(4):104–106, 1994.

[WUM07] Stéphane Weiss, Pascal Urso, and Pascal Molli. Wooki: A Peer-to-Peer Wiki-Based Collaborative Writing Tool. In *Proceedings of the International Conference on Web Information Systems Engineering (WISE)*, pages 503–512, 2007.

[YCM06] Alexander Yip, Benjie Chen, and Robert Morris. Pastwatch: A Distributed Version Control System. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, pages 381—394, 2006.

[You] YouTube - Broadcast Yourself. http://www.youtube.com.

[Zim80] Hubert Zimmermann. OSI Reference Model — The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communication*, 28(4):425–432, 1980.

All web pages cited in this work have been checked in September 2010. However, due to the dynamic nature of the World Wide Web, web pages can change.

CURRICULUM VITÆ

PERSONAL DETAILS

Name: Patrick Mukherjee

Date and Place of Birth: 1st July 1976 in Berlin, Germany
Nationality: German

http://patrick.mukherjee.de

EDUCATION

10/1997 - 10/2004	Studies of Computer Science at Technische Universität Berlin, Germany. Degree: 'Diplom-Informatiker' (corresponds to M.Sc.)
08/1989 – 05/1996	Primary and Secondary Education in Berlin, Germany Degree: Allgemeine Hochschulreife.

PROFESSIONAL EXPERIENCE

since 07/2010	Senior Researcher at the Multimedia Communications Lab of the Department of Electrical Engineering and Information Technology, Technische Universität Darmstadt.
06/2006 - 06/2010	Doctoral Candidate at the Real-Time Systems Lab of the Department of Electrical Engineering and Information Technology, Technische Universität Darmstadt. Topic: *A Fully Decentralized, Peer-to-Peer based Version Control System*
04/2005 - 05/2006	Consultant at Accenture AGS
11/2002 - 04/2004	Student Assistant at the Department PlanT of Fraunhofer Institute for Computer and Software Architecture (FIRST)

ACTIVITIES

since 10/2008	Research Assistants' Representative at the Department Council of the Department of Electrical Engineering and Information Technology, Technische Universität Darmstadt.
11/2006 - 10/2009	Research Assistants' Representative at the Board of Examiners of the Department of Electrical Engineering and Information Technology, Technische Universität Darmstadt.
03/2007 - 02/2008	Founding Member (representing the Research Assistants) of the Joint Commission of the Center for Interdisciplinary Studies (CISP), Technische Universität Darmstadt.
06/2006 - 02/2008	Research Assistants' Representative of the Joint Commission of the Field of Study Information Systems Engineering (iST), Technische Universität Darmstadt.
06/2006 - 02/2008	Research Assistants' Representative at the Board of Examiners of the Field of Study Information Systems Engineering (iST), Technische Universität Darmstadt.
01/2006 - 05/2006	Employee Representative in the Workers' Council at Accenture AGS

04/2007 - 08/2007 Research Assistants' Representative in the appointments committee for a W3 professorship in "Integrated Electrical Systems" at the Department of Electrical Engineering and Information Technology, Technische Universität Darmstadt.

02/2008 - 05/2009 Research Assistants' Representative in the appointments committee for a W3 professorship in "Power Electronics" at the Department of Electrical Engineering and Information Technology, Technische Universität Darmstadt.

REVIEWER ACTIVITES

8th IEEE International Conference on Peer-to-Peer Computing (P2P 2008)

11th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (Models 2008)

Journal of Multimedia Systems, issue 530, 2009 (ISSN: 0942-4962 (print version), ISSN: 1432-1882 (electronic version))

MEMBERSHIPS

Member of the Institute of Electrical and Electronics Engineers (IEEE)

ACADEMIC TEACHING

WS 06/07,WS 07/08, SS 09 Lecture: Software-Engineering - Wartung und Qualitätssicherung (ETiT, Real-Time Systems Lab)

WS 08/09 Practical Course: Bachelor practical course (Computer Science) Topic: NetBrowser (Implementing a generic Network Visualisation)

WS 08/09 Practical Course: Multimedia Communications Lab II (ETiT, Multimedia Communications Lab) Topic: Piki - Reimplementation and Extension as an Eclipse-plugin

WS 07/08 Practical Course: Multimedia Communications Lab II (ETiT, Multimedia Communications Lab) Topic: ASKME - Advanced Peer-to-Peer Instant Messenger as Eclipse Plugin

SS 07, WS 07/08 Seminar: Software System Technology (ETiT, Real-Time Systems Lab)
Topic SS07:
Untersuchung einer Common-API-Empfehlung für strukturierte P2P-Netzwerke
Topics WS07/98:
1. Two Approaches for Standardization of Structured Peer-to-Peer Overlay Protocols
2. Peer-to-Peer Architekturen

SS 07 Practical Course: Peer-to-Peer Infrastructures (Computer Science, Databases and Distributed Systems)
Topic: Piki - A peer-to-peer based wiki engine (initial development)

SUPERVISED DIPLOMA, M.SC., STUDENT, AND B.SC. THESES

Dirk Podolak: *Peer-to-Peer basierte Versionierung mit Schwerpunkt Anwendungsintegration am Beispiel DOORS - Enterprise Architect (Peer-to-Peer based Version Control focussed on the Application Integration DOORS - Enterprise Architect)*; Diploma Thesis (ES-D-0028), 03/2008

Benedikt Antoni: *Lösung von Versionskonflikten beim Verschmelzen von Teilnetzen in einem Peer-to-Peer basierten Versionskontrollsystem (Solving Versions' Conflicts when Melting Partitioned Subnetworks in a Peer-to-Peer based Version Control System)*; Bachelor's Thesis (ES-B-0038), 08/2008

Marwan Hajj-Moussa: *An OSGi Compliant Platform to Reduce the Complexity of Peer-to-Peer Application Development*; Master's Thesis (ES-M-0036), 10/2008

Marcel Queisser: *Zugangs- und Zugriffskontrolle für Peer-to-Peer Netzwerke (Authentication and Access Control for Peer-to-Peer Networks)*; Diploma Thesis (ES-D-0037), 3/2009

Christian Schott: *Inherent Traceability and Change Notification in Distributed Software Development*; Diploma Thesis (ES-D-0006), 04/2009

Lars Almon and Martin Soemer: *Analyse und Bewertung Agiler Entwicklungsmethoden (Analysis and Evaluation of Agile Software Development Methologies)*; Bachelors' Team Thesis (ES-B-0041 and ES-B-0042), 6/2009

Florian Gattung: *Benchmarking and Prototypical Implementation of a Social Knowledge Network*; Diploma Thesis (KOM-D-370) (in collaboration with the Multimedia Communications Lab), 08/2009

Essam Alalami: *Design und Implementierung eines JMI-Repository für Wiki-Engines zur Werkzeugintegration durch modellgetriebene Softwareentwicklung (Design and Implementation of a JMI-Repository for Wiki Engines to enable Tool-Integration through model-driven Softwaredevelopment)*; Master's Thesis (ES-MA-0041), 08/2009

Luciana Alvite: *Development of a Dynamic Flexible Monitored Peer-to-Peer based Social Network Platform using OSGi*; Diploma Thesis (KOM-D-367) (in collaboration with the Multimedia Communications Lab), 11/2009

Robert Vock: *Entwicklung eines Transaktionsprotokolls für Peer-to-Peer basiertes Versionsmanagement (Development of a Transaction Protocol for Peer-to-Peer based Versionmanagement)*; Diploma Thesis (ES-D-0044), 12/2009

Sebastian Schlecht: *Design and Implementation of a Transactional Peer-to-Peer Based Version Control System*; Master's Thesis (ES-M-0043), 12/2009

Czarny, Damian A. und Gebhardt, Alexander: *Entwicklung einer automatisierten Evaluationsplattform für verteilte Anwendungen (Development of a automated Evaluation platform for distributed Applications)*; Bachelors' Team Thesis (ES-B-051 and ES-B-052) (in collaboration with the Multimedia Communications Lab), 12/2009

Jan Schluchtmann: *Untersuchung der Übertragbarkeit von SCM-Techniken auf DHT-P2P-Systeme (Investigation of the transferability of SCM techniques on DHT P2P systems)*; Master's Thesis (ES-MA-0042), 03/2010

March 11, 2011

PUBLICATIONS

JOURNAL ARTICLES

[1] Aleksandra Kovacevic, Sebastian Kaune, Patrick Mukherjee, Nicolas Liebau, and Ralf Steinmetz. Benchmarking Platform for Peer-to-Peer Systems. *it - Information Technology (Methods and Applications of Informatics and Information Technology)*, 49(5):312–319, September 2007.

[2] Kalman Graffi, Aleksandra Kovacevic, Patrick Mukherjee, Michael Benz, Christof Leng, Dirk Bradler, Julian Schroeder-Bernhardi, and Nicolas Liebau. Peer-to-Peer Forschung - überblick und Herausforderungen. *it - Information Technology (Methods and Applications of Informatics and Information Technology)*, 49(5):272–279, September 2007.

CONFERENCE AND WORKSHOP CONTRIBUTIONS

[3] Patrick Mukherjee, Aleksandra Kovacevic, Michael Benz, and Andy Schürr. Towards a Peer-to-Peer Based Global Software Development Environment. In *Proceedings of the 6th GI Software Engineering Conference (SE)*, pages 204–216, February 2008.

[4] Patrick Mukherjee, Aleksandra Kovacevic, and Andy Schürr. Analysis of the Benefits the Peer-to-Peer Paradigm brings to Distributed Agile Software Development. In *Proceedings of the 6th GI Software Engineering Conference (SE)*, pages 72–77, February 2008.

[5] Patrick Mukherjee, Christof Leng, and Andy Schürr. Piki - A Peer-to-Peer based Wiki Engine. In *Proceedings of the 8th IEEE International Conference on Peer-to-Peer Computing (P2P)*, pages 185–186, September 2008.

[6] Patrick Mukherjee, Christof Leng, Wesley W. Terpstra, and Andy Schürr. Peer-to-Peer based Version Control. In *Proceedings of the 14th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 829–834, December 2008.

[7] Kalman Graffi, Sergey Podrajanski, Patrick Mukherjee, Aleksandra Kovacevic, and Ralf Steinmetz. A Distributed Platform for Multimedia Communities. In *Proceedings of the 10th IEEE International Symposium on Multimedia (ISM)*, pages 6–12, December 2008.

[8] Kalman Graffi, Patrick Mukherjee, Burkhard Menges, Daniel Hartung, Aleksandra Kovacevic, and Ralf Steinmetz. Practical Security in P2P-based Social Networks. In *Proceedings of the 34th Annual IEEE Conference on Local Computer Networks (LCN)*, pages 269–272, October 2009.

[9] Kalman Graffi, Christian Groß, Patrick Mukherjee, Aleksandra Kovacevic, and Ralf Steinmetz. LifeSocial.KOM: A P2P-based Platform for Secure Online Social Networks. In *Proceedings of the 10th IEEE International Conference on Peer-to-Peer Computing (P2P)*, August 2010.

Part V

APPENDIX

A ABOUT THE PROTOTYPE'S DEVELOPMENT

The final design of PlatinVC presented in this work is the result of five iterations in the development process. In each iteration we developed a working prototype, which was reprogrammed in the next iteration, based on an improved design. In each iteration we added new features:

- The first prototypehad the basic feature to control the versions of single artifacts only. Named branches and tags were supported as well. The only reused existing component was the communication network layer, free pastry [FP].
- We developed the second prototype on a new basis, which featured link version control (compare to Section 7.3.7).
- Starting with the third version we improved handling of faulty situations, like mechanisms, that handle disturbances like ***network partitioning*** (described in Section 8.6). Large parts of the existing code have been reimplemented, e.g. the replication mechanism has been rewritten from scratch. The additional failure handling mechanisms rose the complexity of the implementation. Thus we based all components on a framework, that handles the components intercommunication and lifecycle (Equinox [equ]).
- In the fourth version we decided to replace our own version control mechanisms for the local version control with another industry proven component (mercurial [Mac]).
- In our final Version, presented in this work, we improved all mechanisms and thus achieved a higher consistency degree.

ERKLÄRUNG

Erklärung laut §9 PromO

Ich versichere hiermit, dass ich die vorliegende Dissertation allein und nur unter Verwendung der angegebenen Literatur verfasst habe. Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

LIST OF FIGURES

Figure 1 Basic principle of centralized version control systems 4
Figure 2 Basic principle of peer-to-peer version control systems 5
Figure 3 Peer to peer based Wiki engine PIKI [MLS08] 11
Figure 4 **P**eer-to-peer based **i**ntegrated **p**roject-support **e**nvironment (PIPE) [MKS08] 14
Figure 5 **P**eer-to-peer based **i**ntegrated **p**roject-support **e**nvironment (PIPE) [MKS08] 15
Figure 6 Globally distributed developers in a GSD scenario 20
Figure 7 The centralized workflow, which is used for distributed as well as for centralized version control systems 36
Figure 8 The integration manager collaboration workflow 36
Figure 9 The lieutenants collaboration workflow 37
Figure 10 Timing diagram of the update process 41
Figure 11 Basic architecture of SCCS and RCS 45
Figure 12 Basic architecture of CVS and Subversion 47
Figure 13 Dynamic (Alice) and Snapshot (Bob) View in ClearCase 50
Figure 14 Basic architecture of dVCSs 52
Figure 15 Sharing versions between participants of a dVCS 53
Figure 16 Alice and Bob share changes in a dVCS 54
Figure 17 Metamodel of Git's version model 56
Figure 18 Metamodel of Mercurial's version model 58
Figure 19 Unstructured and hierarchical peer-to-peer overlay network topologies 62
Figure 20 structured peer-to-peer overlay network topology 64
Figure 21 Basic structure of Pastwatch's repository 66
Figure 22 Basic architecture of PlatinVC 93
Figure 23 Basic Workflow in PlatinVC 95
Figure 24 Conflicts and Resolution in PlatinVC 96
Figure 25 Concurrent updates on a linkdocument 102
Figure 26 Example to clarify which snapshots are pulled. 105
Figure 27 Exemplary history (simplified) 105
Figure 28 High level architecture of PlatinVC 110
Figure 29 Metamodel of the components on a peer 113
Figure 30 Sequence diagram showing basic message exchange using the physical transport layer 119
Figure 31 Messages are resent if no reply is received as shown in this sequence diagram 120
Figure 32 If the direct address of a peer is unknown messages are routed as described by the depicted sequence diagram 120
Figure 33 Similar to the resending presented in Figure 31 routed messages are repeated as well 121
Figure 34 Structure of the exemplary project 123
Figure 35 The history in Alice's local and peer-to-peer module 123
Figure 36 The history in Bob's local and peer-to-peer module after executing the pull operation 124
Figure 37 History (cache marked with dotted lines) after the first pull phase 124
Figure 38 The history in Alice's local and peer-to-peer module after executing the pull operation 125
Figure 39 Activity diagram of the pull operation 126
Figure 40 Activity diagram: Choose the snapshots to which the local repository should be updated to 127

Figure 41 Activity diagram: Get bundle with missing snapshots in the second phase of the pull operation 128
Figure 42 Exemplary history in Bob's peer-to-peer module 130
Figure 43 Exemplary history in Alice's peer-to-peer module 131
Figure 44 History (with cache marked by dotted lines) of Cliff's peer-to-peer module 132
Figure 45 Activity diagram of the global push operation 133
Figure 46 Activity diagram: Open socket during push globally 134
Figure 47 Sequence diagram: Push local snapshots 135
Figure 48 Activity diagram: Prepare to push new history entries 136
Figure 49 Activity diagram describing the actions taken when the neighborhood changes (part 1) 139
Figure 50 Activity diagram describing the actions taken when the neighborhood changes (part 2) 140
Figure 51 The simplified architecture of Pipe 143
Figure 52 The connection options offered by the connection manager (running in Windows) 145
Figure 53 The settings needed by the connection manager, with security enabled (running in Linux) 146
Figure 54 PlatinVC as a stand alone application 148
Figure 55 Settings for PlatinVC (running in OS X) 149
Figure 56 Piki as a stand alone application (running in OS X) 150
Figure 57 Piki integrated in Eclipse (running in Windows) 151
Figure 58 PlatinVC and Askme integrated in Eclipse (running in OS X) 153
Figure 59 Churn execution in our experiments 161
Figure 60 Degree of consistency 162
Figure 61 Percentage of successful operations 163
Figure 62 Freshness of pushed updates (time till available) 163
Figure 63 Time needed to execute the push/pull operation 164
Figure 64 Operation time 165
Figure 65 Size of transferred messages (payload+system messages) 166

LIST OF TABLES

Table 1 Fulfilled requirements (see Section 3.2) 75
Table 2 Guaranteed degree of consistency 81
Table 3 Architectural aspects of the examined VCSs 86
Table 4 Comparison of commit/push and update/pull operation time in Subversion (SVN), Pastwatch [YCM06] and PlatinVC 167
Table 5 Comparison of the features offered by the observed version control systems 168

ACID, 27
Artifact, 16
automatic branches, 99
Automatic Isolation of Concurrent Work, 97

backward delta, 28
based, 39
basis version, 25
binary delta compression, 28
binary diff, 28
bisect, 113
bootstrapping peer, 144
branch, 46
bundle, 132

causal consistency, 42
centralized collaboration workflow, 35
Centralized Version Control System, 45
changeset, 25
check out, 48
cherry picking, 25
Churn, 19
ClearCase, 49
ClearCase Multisite, 51
client-server, 4
client-server communication paradigm, 47
clone, 54
Coherency, 42
commit, 3
concurrency control, 82
concurrent versions system, 46
Consistency, 40
Consistency Degree, 40
Consistency Models, 39
copy-modify-branch, 47
copy-modify-merge, 47
Cryptographic Hash Function, 56
cVCS, 45
CVS, 46

Definition Box, vii
delta compression, 28
derived object, 50
development line, 40
DHT, 65
diff, 28
Distributed Hash Table, 65
Distributed Version Control System, 52
dVCS, 52

encrypt, 66
eventual consistency, 41

failing peer, 19
Flooding, 62
forward delta, 28
framework services, 9
Freshness, 41

Global Knowledge, 118
global push, 106
global repository, 52
Global Software Development, 14
GSD, 14

Hash Collision, 56
Hash Function, 56
Hash Value, 56
head version, 40
heads, 40
Hierarchical Peer-to-peer Overlay Network, 62
history, 23
history cache, 114
history cache entries, 126
hop, 10
hops, 121
hotspots, 68
hybrid concurrency control, 69
hybrid resolving, 82
hybrid voting, 82

indeterministic routing, 140
index, 10
indirect replication, 82
initial version: , 39
integration manager collaboration workflow, 36
iterative routing, 80

JXTA, 62

keep-alive messages, 10

lazy copy, 49
leaving peer, 19
lieutenants collaboration workflow, 36
Link, 13
local commit, 106
local module, 113
lock-modify-write, 46

log head, 66

mainframe communication paradigm, 45
maintainer, 116
maintainer based repository distribution, 61
Merging, 6
Metadata, 114
Metric, 156
Module, 47

NAT, 27
Neighbor Peer, 120
Neighborhood Peers, 10
Network Address Translation, 27
Network Partition, 139
Nomadic Developers, 14

octopus merge, 56
offline messaging, 152
operation-based diff, 70
Optimistic Concurrency Control, 47
outdated, 40
owner path caching, 129

p2p, 5
path, 40, 40
Peer, 5
Peer ID, 9
Peer Proximity, 10
Peer-to-Peer, 5
Peer-to-Peer Application, 9
Peer-to-Peer based Version Control System, 60
Peer-to-Peer Maintenance Mechanisms, 10
Peer-to-Peer Module, 114
Peer-to-Peer Overlay Network, 9
Peer-to-Peer Protocol, 10
Peer-to-Peer Routing Mechanisms, 10
Peer-to-Peer System, 9
Pessimistic Concurrency Control, 46
private key, 66
proactive conflict resolution, 46
public key, 66
Public Key Cryptography, 66
pull, 53
push, 53

Quality Aspect, 155

RCS, 46
reactive conflict resolution, 47
rebase, 54
recursive routing, 80
related versions, 39
replica peers, 65
replica set, 117
replicated repository distribution, 61
repository, 47
revision, 12
revision control system, 46
Robustness, 29
root of trust, 146

Scalability, 29
sequential consistency, 42
serve, 53
SHA-1, 56
Shallow Copy, 49
shifting, 19
sign, 66
Single point of control, 17
Single point of failure, 17
Single point ownership, 17
Snapshot, 23
software product, 93
state-based diff, 70
Structured Peer-to-peer Overlay Network, 64
Subversion, 48
superpeer, 62
SVN, 48
sync, 55
System Load, 29

Three Way Merge, 6
trunk, 46

underlay, 9
unnamed branch, 53
unnamed branches, 96
Unstructured Peer-to-peer Overlay Network, 61
update, 3
user level applications, 9

variant, 12
Version, 12
Version Control System, 3
Version Model, 46
versioned item, 50
versioned object base, 50
virtual global module, 114
Virtual Global Repository, 94
VOB, 50

Working Copy, 48