Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Real-time Conflict Awareness for Distributed Version Control Systems

Master's Thesis

Fabian Gremper
ETH Zürich
fgremper@student.ethz.ch

31.10.2014 — 30.04.2015

Supervised by:
Martin Nordio
H.-Christian Estler
Prof. Bertrand Meyer

ETH Zürich
Chair of Software Engineering

Abstract

In this day and age, most computer programs are written by a team of more than one person. Because software engineering is becoming increasingly distributed, we require smart mechanisms and systems to reduce overhead and stress introduced by a lack of awareness and the need to resolve complicated conflicts.

This thesis introduces a new tool for users of a distributed version control system: CloudStudio is an awareness system for Git, providing information about 'who is making what changes' and possible conflicts through a publicly available API. This allows developers to integrate this information in new ways, e.g. directly as a plugin for common IDEs. CloudStudio can work with all Git repositories and does not require a specific project setup.

CloudStudio divides awareness information into three distinct views: branch view, file view and content view, with each view serving a different purpose and separating the information in a way that is easily understandable for users. Many options and filters can be used in all views to enhance the view for specific needs.

Acknowledgements

I would like to thank Prof. Dr. Bertrand Meyer for giving me the opportunity to realise my master's thesis at the Chair of Software Engineering.

I would like to express my gratitude to my supervisors Dr. Martin Nordio and H.-Christian Estler for their great support throughout my work on this master's thesis.

Contents

Chapter 1

Introduction

1.1 Distributed Software Development

Today's software projects are increasingly distributed across multiple locations over all of the world [1, 2]. This distribution poses new challenges to software development, especially those related to collaboration, as globally distributed development is necessarily collaborative [3].

Many researchers have evaluated the effect of distributed software development [5, 6, 7] and suggested that providing awareness information about who is making what changes may greatly reduce overhead generated by conflict resolution and overall improve the effectiveness of collaboration [3, 4].

Studying the difficulties in communication and collaboration are of utmost importance [9, 2]. The aspects of software development distribution have been researched from many angles [12, 13, 9]. Nordio et al. have researched the impact of contracts in distributed software development to mitigate the risk of misunderstanding software specifications [8], as well as the impact of distribution and time zones on communication and performance in distributed projects [16].

Over the course of several years of teaching "Distributed and Outsourced Software Engineering" (DOSE), Nordio et al. have studied key characteristics in improving collaborative development and have found that the emphasis on API design and development of communication skills are among the leading factors, since at least 30% of the time spent by students in the project has been found to correspond to communication [16, 17].

1.2 Version Control Systems

Version Control Systems (VCS) are widely used in almost all projects with multiple team members.

"In traditional version control systems, there is a central repository that maintains all history. Clients must interact with this repository to examine file history, look at other branches, or commit changes. Typically, clients have a local copy of the versions of files they are working on, but no local storage of previous versions or alternate branches.

Distributed Version Control Systems (DVCS) such as Git and Mercurial have becoming increasingly more popular during the last few years. With DVCS, every user has an entire copy of the repository locally; switching to an alternate branch, examining file history, and even committing changes are all local operations. Individual repositories can then exchange information via push and pull operations. A push transfers some local information to a remote repository, and a pull copies remote information to the local repository." [14]

1.3 Awareness and Conflict Detection

Conventional version control provides a means to collaborate on writing programs, even different subtasks, and merge the changes later. Merging changes can however produce conflicts. To avoid big conflicts, we want to provide the programmer with an awareness system to inform them at the time of writing if another programmer is currently editing parts of the code that may be conflicting with their current work.

Researches have found that interruptions due to insufficient awareness occur frequently for teams of non-trivial size. A large and diverse set of information items has been found to be very important if they are related to the project a distributed software engineer is currently working on [18], while developers often have different preferences regarding the frequency and detail that awareness information should have [3].

1.4 Motivation

With the increasing importance on distributed software development, it has become necessary to construct techniques and tools that can assist programmers to make them more productive in such an environment [11]. Cloud-

Studio is a collaborative development framework proposed by Nordio et al. [10, 25] where the software configuration management, conflict detection, and awareness systems are unitarily conceived and tightly integrated.

This thesis is a proposal at a new software solution, built from ground up, to provide programmers with extensive and relevant awareness information and a mechanism to detect possible conflicts early. I was asked to keep the name CloudStudio for this project. In this section, I will describe some of the differences and aspects that I am trying to improve with my thesis. For this purpose I will refer to my new version as CloudStudio 2.0, and to the previously existing implementation as CloudStudio 1.0.

CloudStudio 1.0 is a web-based IDE that allows users to work, collaborate and run code directly in the browser. Behind the curtains, CloudStudio 1.0 sets up a new Git repository for every project that is used for its intrinsic version control.

The web interface and the functionality of the CloudStudio 1.0 server are deeply intertwined. The system is designed for users to solely work through the web interface and does not provide an API to directly request awareness information from the server, for example to allow inclusion of CloudStudio's awareness features in widely used IDEs, such as EiffelStudio [26] or Eclipse. CloudStudio 2.0 offers an extensive API for this purpose and its web interface communicates directly through this API.

More importantly, CloudStudio 1.0 is dependent on a specific Git repository setup that it creates initially when a new CloudStudio project is created. It is not possible to use any of its functionality with pre-existing Git projects that were not specifically set up in CloudStudio in the first place. While it is possible to retrieve a CloudStudio 1.0 project's Git repository from the backend, perform some work directly on the Git repository and push it back to the server, there are many limitations: e.g. you would be required to follow its internal branch naming conventions and all awareness queries would still have to be done through the web interface.

A big focus of CloudStudio 2.0 is also robustness to possible errors, deviation in repository structures and invalid requests.

With this in mind, CloudStudio 2.0 uses a different approach from its predecessor and has been built from ground up during the course of this master's thesis. The next section, as well as section 2.1.1, will cover the details

and features of the new implementation. From this point on, CloudStudio will refer to this implementation.

1.5 Goals

This project focuses on creating a useful mechanism for users of a distributed version control system to detect possible conflicts early on and provide them with awareness information about who is changing what.

There are several components that are being implemented in order to achieve this:

- A CloudStudio client that needs to run on each developer's machine will gather information about the local Git repository and local working tree and sends relevant information to the CloudStudio server periodically.
- The CloudStudio server will then use the data from all the users running the plugin, as well as the data from a central remote repository (origin) to detect possible merge conflicts that may occur at some point when two or more parties attempt to push their changes. CloudStudio server will also provide extensive awareness information about who is changing what and the current state in which the users are in relation to the central remote repository.
- The CloudStudio server will then provide a well defined API that allows other programs or tools to retrieve this awareness information.
- A web interface will be implemented to access and demonstrate CloudStudio's awareness capabilities.

There are many subgoals to this project:

- The API should be well defined and well documented. In the future, instead of only the web interface, it is conceivable that CloudStudio's awareness information could be also made available directly in the users' IDE through a plugin.
- Awareness information generated by CloudStudio is correct and useful.
- CloudStudio acts as a separate layer on top of Git. Its functionality can be added to existing projects without the need to make any changes to the structure of the Git repository.

- The implementation of all parts should be robust and stable; errors should be dealt with appropriately.
- CloudStudio should be user-friendly and easy to use.

Under section 2.1.1 the features of CloudStudio are listed in detail.

1.6 Related Work

Extensive research in the area of awareness has spawned other tools seeking to raise developers' awareness about the changes introduced by others. The granularity of awareness information varies from tool to tool.

Crystal is a publicly-available tool that uses speculative analysis to make concrete advice unobtrusively available to developers, helping them identify, manage, and prevent conflicts [22].

Syde is a a tool to reestablish team awareness by sharing change and conflict information across developers' workspaces [20]. It uses the abstract syntax tree (AST) to detect conflicts and apply change awareness on the syntax level and was used to investigate conflict detection in a user study [21].

Palantir is an Eclipse plug-in to address direct and indirect conflicts, which arise due to ongoing changes in one artifact affecting concurrent changes in another artifact [19].

FASTDash is an interactive visualisation tool that seeks to improve team activity awareness using a spatial representation of the shared code base that highlights team members' current activities. It provides file-level awareness of the activities in Visual Studio projects [23].

Jazz is an Eclipse plugins that shows simple change awareness by highlighting changed lines, designed to support small, informal teams; anyone can create a team and add or remove members [24].

And of course, the already mentioned original CloudStudio, a web-based framework that shares the changes of developers working on the same project; its real-time awareness system allows for dynamic views on the project by selectively including or excluding other developers' changes [10].

Chapter 2

Design

2.1 Introduction

The following subsections discuss the features that I want CloudStudio to provide and how these features can greatly improve the collaboration experience for users, as well as the approach behind the implementation of CloudStudio.

2.1.1 Features

CloudStudio provides numerous awareness and conflict detection features that users can benefit from. The following is a list of provided awareness information:

- an overview of all users and their relative commit position in relation to the origin. This can show you if users have recently synchronized with the origin, are behind or have made local commits in the meantime.
- the branch that is currently checked out by each user in a project, refered to as the "active branch". This helps users to coordinate their work, especially in big teams with many feature branches for subfeatures.
- the last time since information has been updated for each user, so you never run into the risk of relying on outdated information.
- the last time since CloudStudio has updated its data from the central remote repository (e.g. GitHub).

- from your perspective, what files have been modified by other users, and if so, whether a push by both parties would result in a merge conflict.
- from the perspective of the origin, what files have been modified by which users. If multiple users are working on the same file, it may be appropriate to coordinate further implementations.
- the possibility to stage the same conflicts, but from the perspective from your user in one branch and all other users in another branch. This will highlight changes or conflicts that would occur in the future when these branches are going to be merged together.
- possibility to view either locally committed changes or changes made in the local working tree that have not yet been locally committed. For the latest status information you may want to view uncommitted changes; this may bring up changes that are possibly not meant to be checked in at any point and are just experimental. The latest locally committed changes, however, are going to be pushed to the origin at some point.
- a detailed side-by-side comparison of two versions of a file for two different users, one of which may be the origin.
- a detailed side-by-side merge conflict view for two versions of a file for two different users. In this view, a common ancestor is used as a reference for a three-way merge and will show the same conflicts that would occur when Git tries to merge the files.
- numerous filters that can enhance or narrow down your view to information that is interesting to you: filtering by users allows you to only look at a subset of users; filtering by conflict severity will only show conflicts above a certain threshold.
- grouping awareness information by folders lets you monitor subprojects as a whole.

2.1.2 Approach

CloudStudio can be thought of as an extra layer on top of an existing Git repository setup. There are no requirements for a specific setup in the Git repository and it can work with existing, as well as new projects and provide extensive awareness information and detect possible conflicts. CloudStudio

is divided into two primary parts: a server and a client.

The client is a standalone tool that has to be run in the background by all users in a project that want to benefit of the added awareness information provided by CloudStudio. It will send data to the server periodically and can work with multiple local repositories that are working with the same CloudStudio server. The client provides a graphical user interface that shows its current state and its actions to the user.

The server is hosted at a publicly available hostname and port and can deal with many users and repositories and provides a central, single-login based structure to send and retrieve awareness information to and from. Through a public and well-documented API, this awareness information can be easily integrated with existing IDEs, such as EiffelStudio [26] or Eclipse, or be used in any desired form. For this thesis, a web interface is being implemented and provides access to all the awareness features directly in the browser.

Figure 2.1 shows a typical setup for a single repository: many clients are working on a project using local Git repositories that are synchronised with each other through a central remote repository. Additionally, all users run a CloudStudio client, that will provide the CloudStudio server with the information necessary for it to prepare its awareness information. This way, the two parts function independently from each other and the underlying repository structure is not affected by using CloudStudio. It also means that the central repository can be hosted anywhere (e.g. GitHub) and can still be used with CloudStudio.

Both parts, client and server, are highly customisable through an XML-based configuration file. The configuration is explained in detail in section 3.8.

2.2 Separation of Awareness Views

Awareness information is separated into three different views: branch view, file view and content view. Each view provides distinct information and serves its own purpose; however, some information can be accessed in an overlapping manner, if desired. The views are also built in a way that there is a natural flow for users to navigate from more broad to more detailed information in a repository.

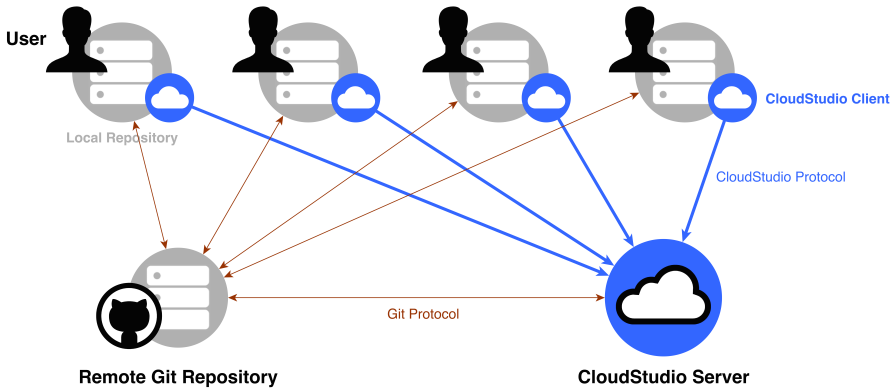

Fig. 2.1: Typical CloudStudio setup for a single repository

The following sections discuss the functionality and purpose of each view in detail.

2.3 Branch Level Awareness

The branch level awareness view is the first view in the navigation sequence for a given repository.

One of the most important awareness features of this view is the visualisation of the local commit state of all users in relation to the origin. In Git, every commit has a unique identifier and all commits are arranged in a graph with each commit having a pointer to a parent commit, making up a directed graph of all commits. While a local repository may not know about all possible commits of all users, we can reconstruct this information using the data sent by the CloudStudio client.

Using this information, the CloudStudio server uses one of the following values to describe each users relationship with the origin: *equal*, *ahead*, *behind*, *fork*, *local branch*, *remote branch*. In case of *ahead*, *behind* or *fork* we are also given a distance. I will demonstrate the meaning behind these values using an example.

Let Figure 2.2 represent the commit history graph for a repository and four users; every commit is shown as a green box with a commit ID. For every user and branch, a reference points to a specific commit, indicating the

commit that we are working on when we are working on a specific branch. The origin also has pointers to a single commit for every branch. The combined commit graph created from all of these individual local commit graphs is seen in Figure 2.3.

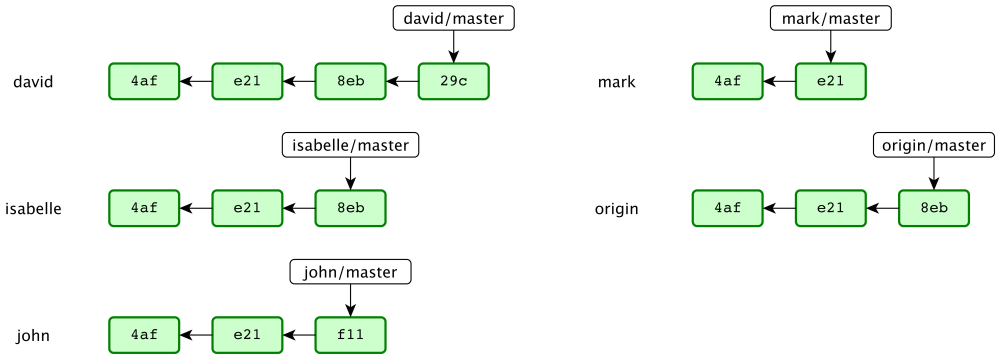

Fig. 2.2: Commit history graph for users in a system

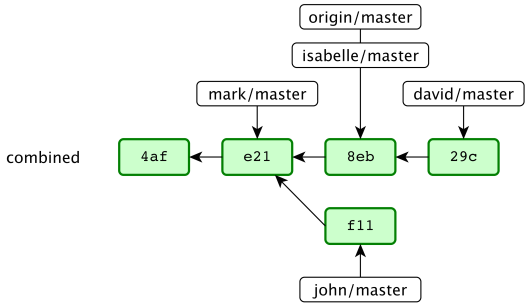

Fig. 2.3: Combined commit history graph

From this graph we can deduct the following relationships:

- Isabelle is *equal* to the origin. Her local repository is up to date and she has not made any further local commits.
- David has made a new local commit after his last repository pull and has not pushed it to the origin yet. He is *ahead* of the origin, with a distance of 1 because he needs to traverse one edge in the graph to reach the commit that the origin is pointing to. David could at this point push his changes to the origin and the origin would simply fast-forward merge, meaning it will add the new commit and point its master branch reference to `29c`.

- Mark is *behind* the origin; he has pulled from the origin when `e21` was the active commit and has not pulled or made a local commit since.
- John is in a *fork*, with a distance of 2, as he has also last pulled from the origin at `e21` but has made a local commit (`f11`). In the meantime someone (either Isabelle or David, given the commit graph) has made a commit (`8eb`) and pushed it to the server. If John wanted to push his changes to the server, a merge between `8eb` and `f11` would be made and stored in a new commit.

The same reference pointers that the example shows for the master branch are also given for all other branches in the system and the calculations behave analogous. If a user has a branch pointer for a branch that isn't on the origin, the relationship is given as *local branch*; for a branch that only exists on the origin but not in a user's local repository, the relationship is *remote branch*.

This awareness feature only deals with locally committed changes and is a useful measure of how up to date local repositories are in relation to the origin. If a user is a *behind*, they should perform a pull. If they are *ahead*, they should push their changes. If they are in a *fork*, it means they started working on a feature while in the meantime someone else pushed their local changes; this case will necessarily result in a merge. *Forks* will occur often, as the only way to avoid them is if only one user is making a change at the time. A low distance can indicate that fewer changes have been made, while a high distance can indicate a big amount of changes.

For each branch in this view, a list of active users is given. If a user has currently checked out a specific branch into their local working tree, it is their "active repository" and it indicates they are working on it. This is very useful to get an overview of what all users are working on, especially if branches have meaningful names (e.g. ticket identifier for feature tickets) and there are many users in a project.

A timestamp shows the last time that the user has updated their state via the CloudStudio client and indicates how recent and accurate the provided information is for each user. This way, users do not falsely rely on outdated information.

2.4 File Level Awareness

As the second step in a usual navigation path through the awareness system, the file level view shows awareness information and possible conflicts for all files in a repository and branch. This is achieved by doing a pair-wise comparison of your version of a file with all the versions of all users in this branch. Alternatively you can also choose to view differences from the origin's point of view.

In its basic form, the file level awareness view compares file checksums to find out whether a file has been modified. If the two file versions that are being compared are not identical, this is presented as a file conflict. It should be noted that non-existing files are treated as empty files for the purpose of this comparison.

2.4.1 View as Origin

If you are viewing from the origin's point of view, a file conflict indicates that the latest version in the remote repository and a user's local file differ; speaking in terms of awareness: all users that are shown up as conflicting have changed their file locally and are probably working on it. If you choose to show uncommitted changes, it will compare the most up to date version of a file from each user's working tree; otherwise it will use the latest locally committed version.

This viewpoint provides useful awareness information to see what users are working on what file. If multiple users show up as a file conflict for the same file, they should coordinate their changes early in order to avoid complicated merge conflicts later on.

2.4.2 View as Yourself

Without the "view as origin" option, your local files are compared directly to all other users'. However, a file conflict is less meaningful here: if you have changed a file, all other users will show up as a file conflict because the file checksums differ. More interesting in this case is to enable the "show conflicts" option.

2.4.3 Show Conflicts

With the "show conflicts" option enabled, the comparison does not directly compare the two file versions, but instead does a three-way comparison by taking the nearest common ancestor of both files from the commit history as the baseline. This emulates the same behaviour that Git would do when performing a merge: Git will select a merge base and try to merge files automatically if different portions of the file have been changed. In this three-way comparison, a merge conflict would by definition occur exactly, if in some part either all three files differ, or if only the base file differs.

For all the files and users that would show up as file conflicts (without this option enabled), this three-way comparison will additionally look for merge conflicts and mark files that would not pass an automatic merge as content conflict. This is very useful, because when two users are working on a file simultaneously but they are working on different parts, no content conflicts are shown. As soon as they are changing the same portions of a file and Git would have trouble merging the files at a later point, a content conflict is displayed. This helps catching merge conflicts very early on and users can arrange their work and take countermeasures before the merge overhead becomes bigger.

2.4.4 Compare to Other Branches

The same functionality can not only be used to show branch internal awareness and conflict information—by specifying a different comparison branch, your (or the origin's) files from your selected branch are compared with all other users' in the comparison branch.

Let's say you are working on the master branch and someone else is working on a feature branch *iss*53 that will have to be merged into the master branch at a later point in time.

By selecting branch comparison with *iss*53 from the viewpoint of the master branch, you can see file and content conflicts that will occur when merging your local master branch state with each other user's *iss*53 branch state. Likewise, by comparing with the master branch from *iss*53's point of view, the comparison will be made between your files from the master branch with all other users' files in the *iss*53 branch.

In the same manner, the comparison functions with the "view as origin"

option enabled: instead of your files from the selected branch, the files from the origin will be compared.

2.4.5 Filters

You can choose to filter your view by only selecting a subset of the users in a project. If the number of users is really big, this helps narrow down your search and display only information relevant to you.

It is also possible to filter files by the severity of conflicts. Selecting "content conflicts" only shows files and users with a conflict type of "content conflicts"; "file conflicts" shows both file and content conflicts, and no filter also shows users where the files are identical.

2.4.6 Grouping

Folders have a grouping mechanism that add up the containing conflicts. If a folder contains at least one content conflict, it will be marked as content conflicting and the users responsible for it will be listed. Likewise file conflicts propagate their conflict type up to the containing folders.

2.4.7 Uncommitted vs. Committed Files

As already mentioned previously, you can always select to work either with locally committed files or uncommitted files directly from the active working tree from all users. Working with uncommitted files has the advantage of always have the latest version of all files, while you are running into the risk that these changes may not be final or only experimental and will never make it into an actual commit. Viewing committed files is more safe in this regard, but may not show very recent changes.

2.5 Content Level Awareness

The content level awareness view allows you to compare two versions of a file side-by-side.

In non-conflict mode, files are compared side-by-side directly and insertions, deletions and modifications are highlighted.

In the conflict mode, the closest common ancestor in the commit history is taken as a base file for a three-way comparison. The comparison is done using the diff3 algorithm that is also used by Git to internally merge files automatically. Sections of the file are matched into blocks; if a block has been changed in all three files or only in the base file, a conflict occurs, because a three-way merge could not automatically decide how to merge these blocks together. These conflicts are highlighted in red, while normal modifications that would be merged automatically by Git are shown in light blue.

The content awareness view allows you to toggle the options to "show uncommitted files", "show conflicts" and "view as origin" directly in place.

Chapter 3

Implementation

3.1 Introduction

CloudStudio consists of three distinct implementation parts at this point: the server, the client and the web interface. The server and client are both written in Java, while the web interface runs in JavaScript.

The client periodically sends data to the server to keep the awareness information of CloudStudio up to date. The server provides a public API to request awareness information directly, as well as a user-friendly web interface that uses and showcases the full capabilities of the API.

The server provides a central login system for its users and allows them to work with Git repositories hosted independently from CloudStudio.

A MySQL database stores all of the data; its details are explained in section 3.7.

3.2 Architecture

3.2.1 Architectural Overview

While in 2.1.2 the entire system was presented from a global point of view, this section deals with the internal architecture of CloudStudio system. Figure 3.1 shows an architectural overview of all entities in CloudStudio.

The CloudStudio server primarily knows about users, repositories and

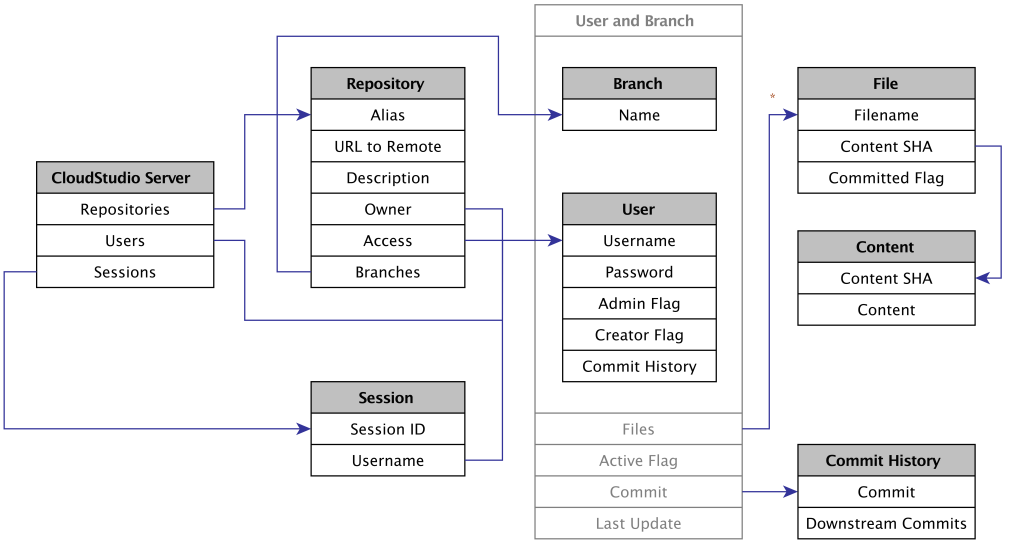

Fig. 3.1: CloudStudio architecture

sessions.

In order to do any sort of data manipulation or awareness requests, a user needs to request a session ID from the server via the *login* API routine. This session ID will then be sent in every subsequent request to the server to verify a user's authentication. A new user can also be created using the *createUser* API command.

A user has a username, a password (that is hashed before storing it), an admin flag (determines whether a user has administrator privileges) and a creator flag (indicating the ability of a user to create new repositories).

Repositories refer to the internal CloudStudio entity of a shared Git repository. A repository has an alias (its internal and unique name in CloudStudio), a description, a URL to a remote repository (e.g. on GitHub), a list of users that have access to it and an owner (who can modify repository specific data, add or remove users, or delete the repository entirely).

For every user and branch in the repository, the information sent by the CloudStudio client is stored individually. On the level of a single repository, this means that every user has one or several branches that contain files, a (partial) commit graph, an active flag (indicating whether a user has cur-

rently checked out this branch into his working tree) and a timestamp when the user has last sent information about this branch to the CloudStudio server.

Files are represented as an object with a filename, a content SHA checksum and a *committed* value. The *committed* value can be one of three values: *committed* (this exact file is currently part of a local commit), *uncommitted* (this is the latest version of a file directly from the working directory) or *both* (the uncommitted and committed files and contents are identical in this branch).

All of the above information is stored in the MySQL database, while the content of the files are stored in the filesystem, named by their SHA. The file object points to its content via the SHA.

A special user named "origin" is automatically added to the server and every repository and represents the central remote repository. It stores the Git information the same way a normal CloudStudio user would.

3.2.2 Logic

One of the key parts to note here is that CloudStudio stores branch and file information for each user and repository separately. Aggregated awareness information is prepared by the server at the time it is requested by the respective API call.

The server does not directly store a combined commit history but only a partial history for every user. More specifically, for every branch in a repository, the transitive set of parent commits of the current branch commit is stored in a list, internally named *downstream commits*. To find the nearest common ancestor commit to calculate a merge conflict, we can simply go through the list of downstream commits of two users and choose the one with the smallest combined distance from the respective branch commit.

Every CloudStudio repository can have a central remote Git repository URL attached. CloudStudio needs to fetch its information regularily for two main reasons:

1. Retrieving the commit graph in order to find the relative position for the branch awareness view.

2. The CloudStudio client only sends commit data (such as files and their content) for commits that have a local branch reference point to them. For conflict detection, files in the merge-base commit are directly looked up from the origin through JGit. Since we assume that users synchronise their repositories through the specified remote repository, this always works, because the merge-base commit necessarily needs to have been pushed to the origin at some point.

As much of the logic as possible has been implemented directly as SQL queries, which positively affects the performance. This works especially well for branch and file awareness. For content awareness, a lot of calculations are performed directly in Java: file contents need to be looked up through JGit and comparison is implemented using the respective diff and diff3 algorithms.

3.2.3 Access Control

CloudStudio provides a central login structure, which can be used by many users collaborating on different shared projects. Each repository has a designated owner who has the rights to add or remove people to the project, change its metadata, elect a new owner, or remove the repository altogether.

Administrators can manage users and their privileges, as well as perform any actions that a repository owner or normal user could. A "creator" flag for each user indicates whether or not they have the privilege to create new repositories on CloudStudio. In the server configuration, you can enable or disable to set this flag by default for new users. In some closed environments, e.g. teachers set up projects for students and add them to the project, it may be preferred that not all users can create new repositories on CloudStudio.

3.2.4 Folder Structure

The source files are divided into 4 folders at the root level: *CSClient* contains the client classes, *CSServer* contains the server classes and the web interface, *CSCommon* contains classes shared by client and server, and *CSTesting* contains the JUnit tests used to verify CloudStudio's correctness.

3.3 Client

The client is responsible for periodically sending information for all the local Git repositories that are being monitored by CloudStudio. All users in

Class	Description
ClientMain	This is the main class. It initiates reading the configuration file, launching the GUI, and periodically reading the local Git repositories and sending the gained information to the server.
ClientGUI	Renders the GUI elements using Swing.
HttpClient	Communicates with the CloudStudio server API using an HttpUrlConnection.
RepositoryReader	Reads a local repository using JGit and retrieves information relevant to CloudStudio's awareness capabilities.
ClientConfig	Container for configuration data.
RepositoryInfo	Container for repository data.

Tab. 3.1: Client classes

a repository should use the CloudStudio client; however if only a subset of the users use it, awareness information is still prepared by the CloudStudio server.

The client is written in Java and uses JGit [15], a Java library to read and manipulate local Git repository information. It periodically retrieves the local commit graph structure, branch references, and files and their contents, for both uncommitted and committed files, and then sends relevant information to the CloudStudio server. This is done using a single API call *localState* and the exact details can be found in the API Reference. Table 3.1 shows the client's Java classes and quickly describes their function. The code for all individual classes is commented throughout. For detailed information, have a look at the source code.

Configuration of the client is done using an XML file. By default it looks for a file named `config.xml` in the same folder; alternatively, you can specify a config file location as the first command line parameter when running the client. In order to successfully use the client, you need to first create a CloudStudio login and specify it in your configuration file. Configuration management of client and server is explained in detail in section 3.8.

To facilitate the users' experience, a graphical user interface (seen in Fig. 3.2), using Swing, keeps you informed using a virtual traffic light, indicating the state of the client, a progress bar to indicate the next time that information is pushed to the server, a force update button and a log view for more detailed information. A green light means everything is functioning correctly, yellow indicates that some sort of error has occurred at some point but the system is still functioning (see the log view in the GUI for details), and a red light means that a hard error has occurred that the system could not recover from. At this point, the graphical user interface cannot be used to manipulate the configuration of the client.

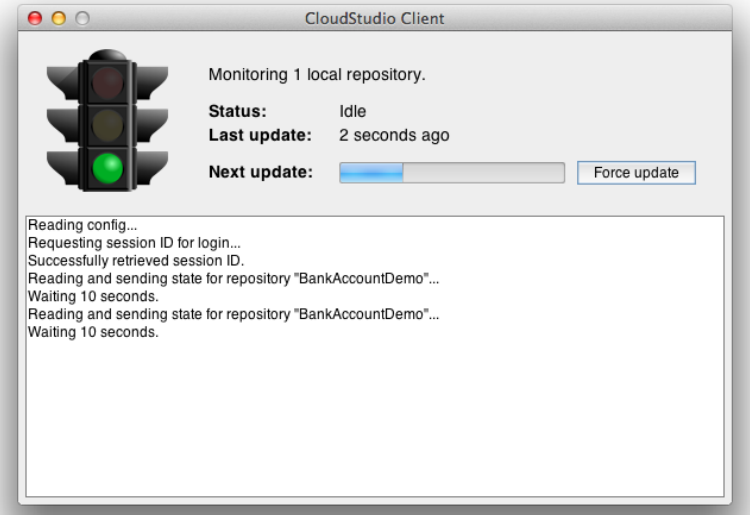

Fig. 3.2: Screenshot of the CloudStudio client

3.4 Server

The CloudStudio server is written in Java and is the core of the system. It reads, stores and processes information necessary for awareness requests and conflict detection. There are two main services running on the CloudStudio server: an HTTP server and a service to periodically update information from the central remote Git repositories used by CloudStudio projects.

The HTTP server implements the multi-threaded `com.sun.httpserver` and two separate request handlers deal with API requests and web interface requests on the same port. The class ApiHttpHandler is responsible for calls to URLs prefixed with `/api/` and provides a public interface for the main CloudStudio server functionality. Any other URLs will be directed to the WebInterfaceHttpHandler, which is mostly just a static web server; the actual web interface runs in JavaScript and uses the API directly. Details about the implementations of the web interface follow in the next section.

PeriodicalAllOriginUpdater is the service to periodically update the remote repository information for all CloudStudio projects. Behind the cur-

Class	Description
ApiHttpHandler	Handles HTTP exchanges for API requests.
ContentConflictGitReader	Finds the common ancestor needed to find conflicts and do three-way comparisons.
DatabaseConnection	Performs operations on the database.
DatabaseConnctionPool	Accesses and returns a data source from C3P0 database connection pool.
OriginUpdater	Updates remote Git repository information for a single repository and writes information into the database.
PeriodicalAllOriginUpdater	Periodically calls OriginUpdater with all the remote repositories used by CloudStudio projects.
ServerConfig	Reads the configuration file and returns individual parameters.
ServerMain	Main class that reads the config, sets up the origin updater and starts the HTTP server.
SideBySideDiff	Prepares a side-by-side comparison JSON object for two files.
SideBySideThreeWayDiff	Prepares a side-by-side compairson JSON object for a three-way comparison.
SqlQueryReader	Reads and caches SQL queries stored in external files.
WebInterfaceHttpHandler	Handles HTTP requests to the web interface.
ProcessWithTimeout	Helper class to allow a timeout for Process executions.
ParameterFilter	Helper class to parse HTTP GET and POST parameters.

Tab. 3.2: Server classes

tains, it clones the given Git repositories and reads and stores them in the same manner as a regular client would. It uses the special user account "origin", which is added to all projects by default. Table 3.2 shows an overview of the server's Java classes. The code for all individual classes is commented throughout. For detailed information, have a look at the source code.

All CloudStudio data is stored in a MySQL database, accessed either directly through the JDBC driver or using the C3P0 database connection pool, depending on the server settings.

The server is greatly customizable through a configuration XML file, described in section 3.8.

3.5 Web Interface

CloudStudio has a web interface that allows you to view awareness information from the browser. The web interface runs on the same server and port as the API. The logic of the web interface runs in JavaScript and on the client side; an approach that is popular with many web services nowadays. The server serves as a static webserver, and awareness data is fetched directly via the CloudStudio API.

The CloudStudio web interface uses EJB, a light-weight templating en-

gine, and jQuery, a fast, small, and feature-rich JavaScript library that makes thinks like HTML document manipulation, event handling, and Ajax much simpler.

The web interface was designed in a modern way that is user intuitive and easy to understand. The functionality of the web interface is explained in detail in section 4.2.

3.6 API

The CloudStudio API exposes an interface to access and manipulate CloudStudio resources. All CloudStudio resources are accessed and manipulated in a similar way. Requests to the CloudStudio API have to use either the GET or POST method. GET requests are used for functions that do not change the state of the database. POST requests are used for functions that make changes to the database.

The content type of requests to the CloudStudio API must be set to `application/x-www-form-urlencoded`. The response has the content type `application/json`. This asynchronism allows to provide parameters for both GET and POST requests similarly and still retrieve comprehensive JSON objects, and is used by many widely used APIs (e.g. SoundCloud). The entire API documentation can be found in the Appendix of this thesis or directly on the project's GitHub page.

3.7 Database

The back-end for all data stored in the system is a MySQL database. The database is accessed using the JDBC driver directly, or using the C3P0 connection pool framework, depending on the configuration settings. Figure 3.3 shows the database tables and their primary (indicated as *PK*) and foreign keys (indicated as arrows). Table 3.3 explains the functionality of each table.

3.8 Configuration Management

3.8.1 Client Configuration

In order to run the client JAR, you need to have a configuration file called `config.xml` in the same directory. Alternatively, you can also specify the

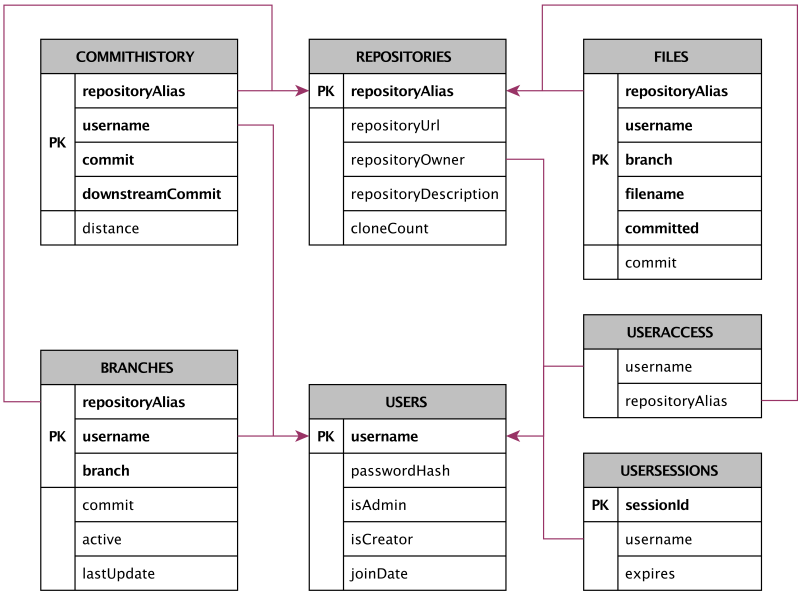

Fig. 3.3: CloudStudio's database setup

Table	Functionality
USERS	Every entry represents a CloudStudio user
REPOSITORIES	Every entry represents a CloudStudio repository
USERSESSION	Every entry represents a session
USERACCESS	Which user has access to a repository (n:n)
BRANCHES	Branches for a user and repository
FILES	Files for a user, repository and branch
COMMITHISTORY	Commit history for a user and repository

Tab. 3.3: Database tables

path to a config file as the first parameter. A sample configuration file can be seen in Fig. 3.4.

Specify your *username* and *password* that you previously created for CloudStudio using its API or the web interface. Your user must have been added to the repository on CloudStudio.

Under repositories you can list multiple repositories that you want to monitor with CloudStudio. The *repositoryAlias* is the name that a given repository has on CloudStudio and the *localPath* is the folder where you locally cloned your Git repository into. Specify a time interval in seconds as the *resubmitInterval*, indicating how often the client sends data to the CloudStudio server.

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
    <username>John</username>
    <password>burgers</password>
    <serverUrl>http://cloudstudio.ethz.ch:7330</serverUrl>
    <repositories>
        <repository>
            <alias>RepositoryAliasOnCloudStudio</alias>
            <localPath>/path/to/your/local/repository</localPath>
        </repository>
    </repositories>
    <resubmitInterval>300</resubmitInterval>
</config>
```

Fig. 3.4: Sample client configuration

3.8.2 Server Configuration

Figure 3.5 shows a sample configuration. Table 3.4 explains what the individual parameters do. After setting up the configuration file, you need to run `SQLInit.sql` to initialize the database (MySQL), before starting the server.

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
    <serverPort>7330</serverPort>
    <dbDriverClass>com.mysql.jdbc.Driver</dbDriverClass>
    <dbJdbcUrl>jdbc:mysql://localhost/cloudstudio</dbJdbcUrl>
    <dbUser>dbadmin</dbUser>
    <dbPassword>1234</dbPassword>
    <useDatabasePool>true</useDatabasePool>
    <dbMinPoolSize>5</dbMinPoolSize>
    <dbAcquireIncrement>5</dbAcquireIncrement>
    <dbMaxPoolSize>20</dbMaxPoolSize>
    <dbMaxStatements>180</dbMaxStatements>
    <fileStorageDirectory>path/to/filestorage</fileStorageDirectory>
    <originStorageDirectory>path/to/origins</originStorageDirectory>
    <passwordSalt>GXSBML0EGj0MfqPzsznUCkK8ENP3lmOX</passwordSalt>
    <enableOriginUpdate>true</enableOriginUpdate>
    <originUpdateInterval>300</originUpdateInterval>
    <createAdminUser>true</createAdminUser>
    <giveCreatorPrivilegesOnSignUp>true</giveCreatorPrivilegesOnSignUp>
</config>
```

Fig. 3.5: Sample server configuration

3.9 Testing and Correctness

The Eclipse plugin EclEmma [27] was used to create coverage reports and view the line coverage directly in the workbench. EclEmma is a free code

Setting	Description
serverPort	Port for the HTTP server hosting the API and the Web Interface
dbDriverClass	JDBC driver
dbJdbcUrl	Database URL
dbUser	Database username
dbPassword	Database password
useDatabasePool	Enable C3P0 database pooling (*true*/*false*)
dbMinPoolSize	C3P0: minimum pool size
dbAcquireIncrement	C3P0: acquire increment
dbMaxPoolSize	C3P0: maximum pool size
dbMaxStatements	C3P0: maximum database statements
fileStorageDirectory	The database only stores file hashes. The file contents to the hashes are stored in this directory.
originStorageDirectory	A clone of the remote repository is stored in this directory for all projects.
passwordSalt	Salt for the password hash
enableOriginUpdate	Periodically fetch all remote repositories (*true*/*false*)
originUpdateInterval	How often to update remote repositories (in seconds)
createAdminUser	Create an administrator with username "Admin" and password "1234" if it doesn't exist on server start (*true*/*false*)
giveCreatorPrivilegesOnSignUp	Automatically give repository creation privileges when a new user is created (*true*/*false*)

Tab. 3.4: Server configuration parameters

coverage tool, based on the JaCoCo library. A high code coverage indicates that the code has been more thoroughly tested and there is a lower chance of software bugs than in a program with low code coverage [28]. The coverage report created by EclEmma can be seen in Figure 3.6.

To verify CloudStudio's correctness, I implemented two different types of JUnit [30] tests: *class tests* for client and server verify the correct behaviour for a given class and are named after the class that is being tested, e.g. ApiHttpHandlerTest for ApiHttpHandler; *combination tests* run longer scenarios of a typical CloudStudio workflow and assert correct behaviour throughout.

The CloudStudio client and server use the Apache Log4j [29] framework to create log output; it can be customised by editing the `log4j.xml` file. Log files can as well be used to view and ensure the correct behaviour of the code.

3.10 Build and Run

Follow the following steps to build the CloudStudio server and client locally:

1. *Clone the project.*
   `git clone https://github.com/fgremper/CloudStudio.git`

2. *Import into Eclipse.*

Class	Instruction			Branch		Line	
	Coverage	Miss	Cov.	Miss	Cov.	Miss	Cov.
CLIENT							
ClientConfig	100%	0	8	0	0	0	2
ClientConfigReader	88%	27	197	12	14	0	33
ClientGUI	87%	58	401	21	7	20	87
ClientGUI.new ActionListener() {...}	50%	3	3	0	0	2	2
ClientGUI.new Thread() {...}	64%	4	7	0	0	2	5
ClientGUI.new WindowAdapter() {...}	50%	3	3	0	0	2	2
ClientMain	53%	170	195	42	16	29	37
ClientMain.new Runnable() {...}	100%	0	5	0	0	0	4
HttpClient	95%	12	249	2	6	4	54
RepositoryInfo	100%	0	9	0	0	0	4
SERVER							
ApiHttpHandler	84%	207	1080	80	108	29	256
ContentConflictGitReader	88%	53	397	14	20	16	64
DatabaseConnection	91%	195	1902	39	89	22	473
DatabaseConnectionPool	0%	68	0	2	0	18	0
OriginUpdater	92%	16	178	3	3	4	36
PeriodicalAllOriginUpdater	0%	84	0	2	0	28	0
ServerConfig	81%	29	122	11	7	2	24
ServerMain	0%	101	0	4	0	27	0
SideBySideDiff	59%	139	200	16	14	29	45
SideBySideThreeWayDiff	86%	100	596	14	48	8	100
SqlQueryReader	91%	5	48	0	4	3	10
WebInterfaceHttpHandler	95%	9	161	11	13	0	31
COMMON							
ParameterFilter	78%	42	145	8	14	14	33
ProcessWithTimeout	81%	6	26	0	0	4	11
RepositoryReader	98%	11	563	4	32	1	108

Fig. 3.6: Test coverage using EclEmma

Import the 4 folders CSClient, CSServer, CSCommon and CSTesting as existing Eclipse projects. (Open Eclipse and go to *File* → *Import* and select *Existing Projects into Workspace*.)

3. *Build JAR*
   In Eclipse, go to *File* → *Export* → *Java* → *Runnable JAR file*. Under *Launch configuration*, select ClientMain to build the client JAR. To build the server JAR, select ServerMain. Under *Library handling*, select *Package required libraries into generated JAR*. Select the export destination and click Finish.

4. *Run*
   Run the client: `java -jar CSClient.jar`
   Run the server: `java -jar CSServer.jar`

Chapter 4

User Guide

4.1 Setup and Run Client

The fastest way to get started with CloudStudio is to use a precompiled client JAR directly from GitHub.

- Download the latest `CSClient.jar` directly from: `http://github.com/fgremper/CloudStudio`
- Go to `http://cloudstudio.ethz.ch:7330/` and create a new account by clicking "Sign up" in the top right corner and providing a new username and password.
- Create a new `config.xml` file in the same directory as the client JAR you downloaded and paste in the setup configuration.
- Replace the username and password with your username and password you just created.
- If you want to work with an existing CloudStudio project, provide its repository alias and the path to your local Git repository. Make sure the repository owner added you to the repository access list.
- If you want set up a new CloudStudio project, click "Create repository" in the repository overview and provide an alias, description and possible an URL to a remote repository. Use the repository alias in your configuration file.
- You can use the client to monitor multiple repositories with the same CloudStudio user account.

4.2 Using the Web Interface

This section quickly guides you through the different views in the CloudStudio web interface and shows you how to use them.

4.3 Login

This is the initial view of the CloudStudio web interface. Log in with your credentials or choose to sign up for a new account in the top-right corner.

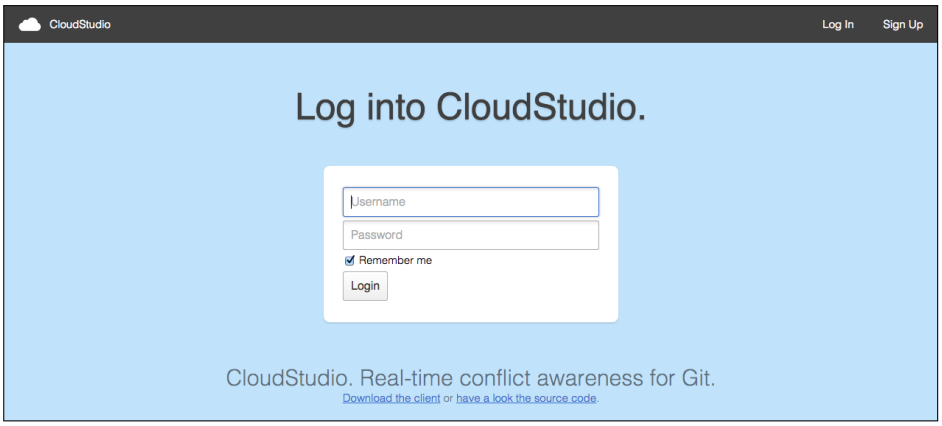

Fig. 4.1: Login view

4.4 Repository Overview

The *Repository Overview* is the main view after you log into the web interface. It provides a list of all repositories you have access to.

If you have administrator privileges or are the repository owner, you can add or remove users from the project, as well as change the repository URL (path to a remote, e.g. GitHub) or description, by clicking the *Edit* button in the top-right corner of a repository. If you have repository creation privileges, you can create a new CloudStudio repository by clicking the *Create repository* button at the top.

Click on a repository to step into the branch level awareness view.

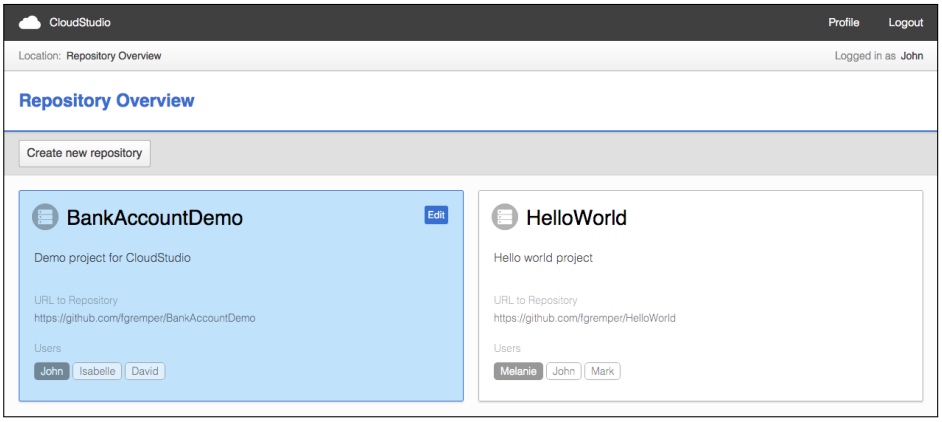

Fig. 4.2: Repository overview

4.5 Branch Level Awareness

In this view, we are interested in the branch pointers of all users in relation to the branch pointers of the origin.

Assume we are looking at the master branch. If a user's master branch reference points to the same commit as on the origin, then the user's relationship is given as "Up to date". If the user has made a new commit locally but has not pushed it to the server yet, the user is displayed under "Made new commits". If someone else has pushed a new commit to the origin after our user has last pulled from the origin, the user is listed under "Behind origin".

Local branches that have not been pushed to the origin are given as "Local branch"; if a user has not fetched a given branch, it is displayed as "Remote branch".

Users listed under "Working On This Branch" have currently checked out this branch locally. The time since the last time the CloudStudio Client has been run is given next to the username in brackets. In the top-right corner you can also see when information from the central repository has last been retrieved.

Clicking on a branch takes you to the file level awareness view for a given branch.

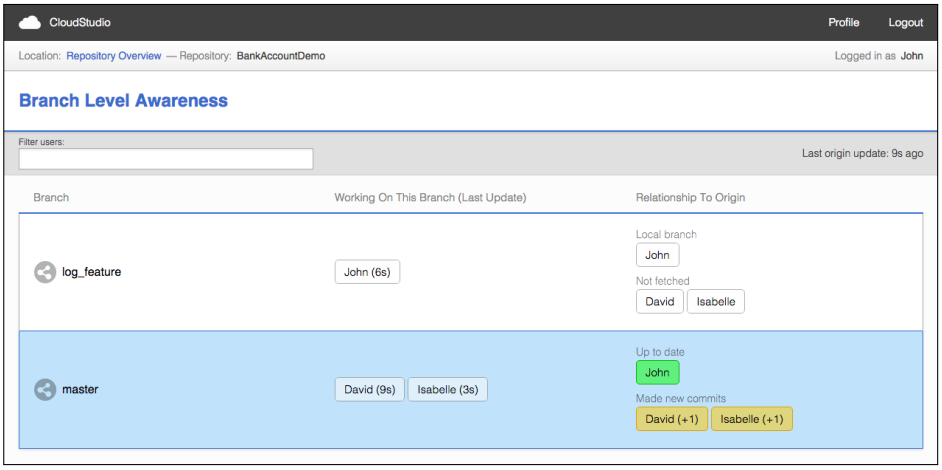

Fig. 4.3: Branch level awareness

4.6 File Level Awareness

For each file in a branch, file conflicts between you and every other user are displayed. A file conflict occurs when two files aren't identical and is highlighted in yellow. In red, content conflicts are displayed—a content conflict occurs if a merge of two files would result in a merge conflict. This feature uses a three-way comparison approach with a common ancestor of the two files as a merge-base in order to mimic the functionality of a Git merge.

By selecting the *view as origin* at the top, instead of comparing your files to those of all others, the files of the origin are taken as the base for comparison. Various filters let you selectively show or hide users, enable or disable the content conflict feature, and view the latest uncommitted versions of files or deal with contents of files that have already been locally committed.

Folders act as groups and the conflict status of files are propagated upwards. If at least one file in a folder is red, the parent folder becomes red and the users responsible for the content conflict are shown. Likewise, if there are file conflicts in a folder but no content conflicts, the enclosing folder becomes yellow. This functionality is especially useful if your project is setup in subfolders for different features.

The comparison branch refers to the branch that you compare your files to. This is useful if you know that the branch you're working in is going to be merged into another branch soon and you want to see what conflicts would occur.

By clicking on a user you can step into the content awareness view, that lets you view the differences between the two of you.

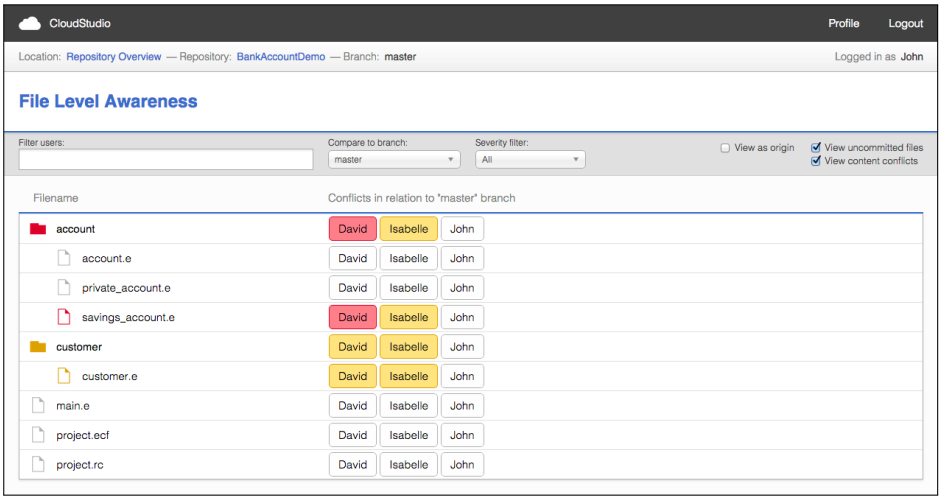

Fig. 4.4: File level awareness

4.7 Content Awareness

In this view, two files are compared side-by-side. Without checking the *show conflicts* options, files are compared directly to each other and the differences are highlighted.

When you choose to *show conflicts*, a common ancestor of the two files is used as a merge-base to create a three-way comparison. Per definition, a conflict occurs when three blocks all differ or only the ancestor differs. If there is at least one conflict, we say that there is a *content conflict* for this file. As before, you can still switch between viewing the committed and uncommitted files directly.

4.8 User Management

Administrators can view users, manage their privileges, or delete their account altogether.

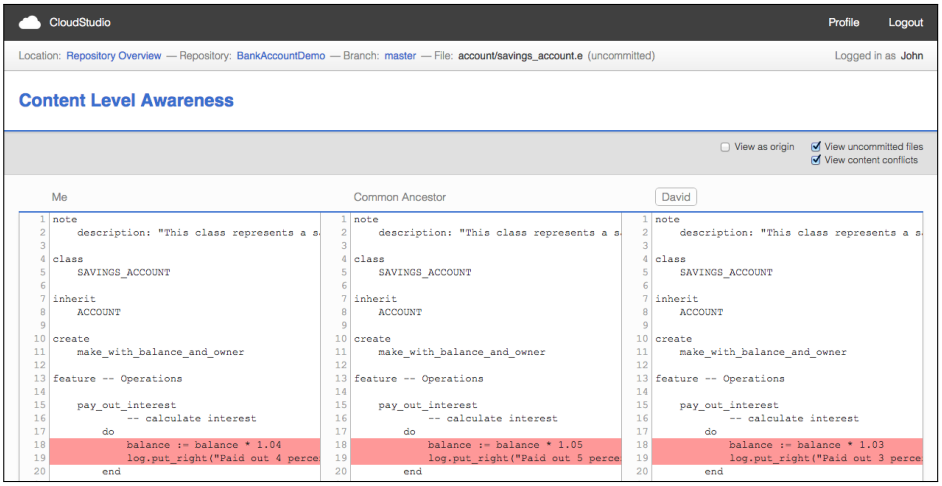

Fig. 4.5: Content level awareness

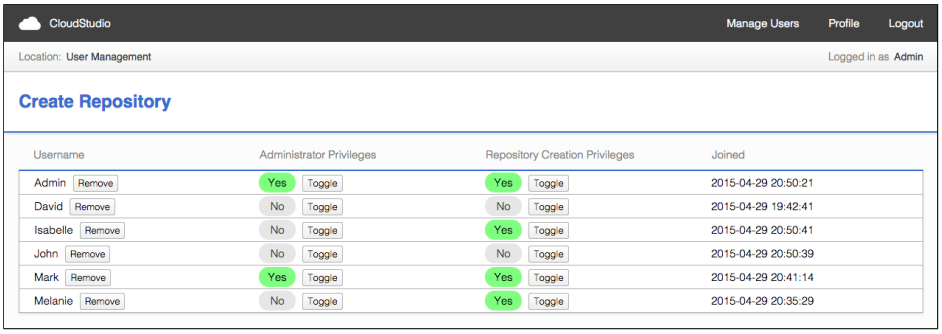

Fig. 4.6: User management

4.9 Create Repository

This view lets you set up a new repository. The URL provided as remote repository will be periodically fetched by the CloudStudio server to make sure its data will be up to date.

4.10 Edit Repository

As an administrator or repository owner, this view lets you make modifications to its metadata, add or remove users, set a new repository owner or delete the repository altogether.

Chapter 5

Future Work

The functionality integrated by this thesis can be easily extended.

While CloudStudio performed sufficiently in test projects, there are many steps that can be taken to improve its overall performance. To load the file level awareness view with conflict detection, a lot of file comparisons have to be made and files (common ancestors) have to be looked up through JGit. The results of operations like these could be cached to speed up the service.

CloudStudio offers an API that opens up the possibility to write all sorts of plugins that benefit from its awareness and conflict detection capabilities. Aside from using it directly through the web interface, future work can include writing plugins that directly display awareness information in a programmer's preferred IDE.

As of this time, the client sends its entire information to the server every periodical update. It was a conscious decision not to send incremental one-way updates from the client to preserve the stability of the system. However, it is conceivable to implement a delta update function where client and server negotiate what information has to be sent in order to lower the bandwidth requirements.

Git treats every commit as a snapshot, and as such is unaware of file renaming. It does however use heuristics to calculate the likelihood of a file rename given the similarity of two files. CloudStudio has not yet implemented any heuristics to detect file renames and would greatly benefit from doing so.

The Chair of Software Engineering at ETH Zurich has been teaching a

"Distributed and Outsourced Software Engineering" (DOSE) course for several years to prepare students for new challenges in a distributed development environment. [16, 17] CloudStudio can be used as a means of collaboration and to heighten the sense of awareness in student projects, by both students and the teaching assistants.

Chapter 6

Conclusions

Software engineering is becoming an increasingly distributed activity. Teams are spread out over all of the world and new problems arise, such as the lack of communication and awareness information, which may disrupt progress and jeopardise efficiency and timeliness [3].

CloudStudio proposes a new mechanism for making awareness information available and detect conflicts early on. One of the key features of CloudStudio is the opening of its functionality to developers by providing a public and well-documented API. This allows for integration of CloudStudio's awareness information into new services and common IDEs. CloudStudio acts as a separate layer on top existing Git projects and as such can be added at any time, while no specific structure of the Git repository is required.

For the implementation of CloudStudio, numerous feature requirements have been set from the start, listed under 2.1.1. This has been done in order to make sure that the information generated by CloudStudio is useful and accurate. The criteria for success have been specified in a project plan before starting the thesis and have been tightly followed. Furthermore, many new features and a sophisticated web interface have been added.

Among the big challenges of the thesis were the distributed nature of the project and making sure all the individual parts work together smoothly, studying the structure of Git and coming up with an awareness system that is useful, realisable and uses the available information from Git repositories. Also, the focus on stability and good error handling required a lot of testing and fixing small bugs.

Many extensions to the existing version of CloudStudio are conceivable, some of which are listed in Chapter 5. CloudStudio offers an ideal platform for the "Distributed and Outsourced Software Engineering" (DOSE) course, allowing students to experiment with new sources of awareness information and hopefully improving the workflow of all participants.

Bibliography

[1] Martin Nordio, Roman Mitin and Bertrand Meyer. **Advanced Hands-on Training for Distributed and Outsourced Software Engineering**, In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ACM. 2010.

[2] E. Carmel. **Global software teams: collaborating across borders and time zones**. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[3] **Awareness and Merge Conflicts in Distributed Software Development**. H.-Christian Estler, Martin Nordio, Carlo A. Furia, Bertrand Meyer, In Proceedings of the 9th International Conference on Global Software Engineering (ICGSE) (Yuanfang Cai, Jude Fernandez, Wenyun Zhao, eds.), IEEE Computer Society, 2014.

[4] Y. Brun, R. Holmes, M. Ernst, and D. Notkin. **Proactive detection of collaboration conflicts**. In ESEC/FSE, pages 168-178. ACM, 2011.

[5] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy. **Does distributed development affect software quality? An empirical case study of Windows Vista**. In Proceedings of the 31st International Conference on Software Engineering, ICSE ?09, pages 518-528, Washington, DC, USA, 2009. IEEE Computer Society.

[6] J. Herbsleb and a. Mockus. **An empirical study of speed and communication in globally distributed software development**. IEEE Transactions on Software Engineering, 29(6):481-494, June 2003.

[7] J. A. Espinosa, N. Nan, and E. Carmel. **Do Gradations of Time Zone Separation Make a Difference in Performance? A First Laboratory Study**. In International Conference on Global Software Engineering (ICGSE 2007), pages 12-22. IEEE, Aug. 2007.

[8] Martin Nordio, Roman Mitin, Bertrand Meyer, Carlo Ghezzi, Elisabetta Di Nitto and Giordano Tamburelli. **The Role of Contracts in Distributed Development**. In proceedings of Software Engineering Advances For Offshore and Outsourced Development (SEAFOOD), Lecture Notes in Business Information Processing 35, Springer-Verlag, 2009.

[9] M. Nordio, H.-C. Estler, B. Meyer, J. Tschannen, C. Ghezzi, and E. D. Nitto. **How do distribution and time zones affect software development? A case study on communication.** In Proceedings of the IEEE International Conference on Global Software Engineering (ICGSE 2011). IEEE, 2011.

[10] **Unifying Configuration Management with Awareness Systems and Merge Conflict Detection**. H.-Christian Estler, Martin Nordio, Carlo A. Furia, Bertrand Meyer, In 22nd Australasian Software Engineering Conference (ASWEC), IEEE, 2013.

[11] **Collaborative Debugging**. H.-Christian Estler, Martin Nordio, Carlo A. Furia, Bertrand Meyer, In 8th International Conference on Global Software Engineering (ICGSE), IEEE, 2013.

[12] J. A. Espinosa, N. Nan, and E. Carmel. **Do gradations of time zone separation make a difference in performance? A first laboratory study**. In Proceedings of the IEEE International Conference on Global Software Engineering (ICGSE 2007), pages 12-22. IEEE, Aug. 2007.

[13] H.-C. Estler, M. Nordio, C. A. Furia, B. Meyer, and J. Schneider. **Agile vs. structured distributed software development: A case study**. In Proceedings of the 7th International Conference on Global Software Engineering. IEEE, 2012.

[14] **Analysis of Git and Mercurial**. https://code.google.com/p/support/wiki/DVCSAnalysis

[15] **JGit**. http://eclipse.org/jgit/

[16] Martin Nordio, Carlo Ghezzi, Bertrand Meyer, Elisabetta Di Nitto, Giordano Tamburrelli, Julian Tschannen, Nazareno Aguirre, Vidya Kulkarni. **Teaching Software Engineering using Globally Distributed Projects: the DOSE course**, In Collaborative Teaching of Globally Distributed Software Development - Community Building Workshop (CTGDSD), ACM, 2011.

[17] Martin Nordio, Roman Mitin and Bertrand Meyer. **Advanced Hands-on Training for Distributed and Outsourced Software Engineering**, In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ACM. 2010.

[18] K. Dullemond and B. van Gameren. **What distributed software teams need to know and when: An empirical study**. In ICGSE, pages 61-70, 2013.

[19] A. Sarma, G. Bortis, and A. van der Hoek. **Towards supporting awareness of indirect conflicts across software configuration management workspaces**. In ASE, pages 94-103. ACM, 2007.

[20] L. Hattori and M. Lanza. Syde. **A tool for collaborative software development**. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, pages 235-238. ACM Press, 2010.

[21] L. Hattori, M. Lanza, and M. D'Ambros. **A qualitative analysis of preemptive conflict detection**. Technical Report 2011/05, University of Lugano, Sept. 2011.

[22] Y. Brun, R. Holmes, M. Ernst, and D. Notkin. **Proactive detection of collaboration conflicts**. ESEC FSE, Szeged, Hungary, 2011.

[23] J. Biehl, M. Czerwinski, G. Smith, and G. Robertson. **Fastdash: a visual dashboard for fostering awareness in software teams**. In Proceedings of the SIGCHI conference on Human factors in computing systems, CHI ?07, pages 1313-1322, New York, NY, USA, 2007. ACM.

[24] S. Hupfer, L.-T. Cheng, S. Ross, and J. Patterson. **Introducing collaboration into an application development environment**. In Proceedings of the 2004 ACM conference on Computer supported cooperative work, CSCW ?04, pages 21-24. ACM, 2004.

[25] **Automatic Version Control System for Distributed Software Development**. Sandra Weber, ETH Zurich, 2012

[26] **EiffelStudio**. https://www.eiffel.com/eiffelstudio/

[27] **EclEmma**. http://www.eclemma.org/

[28] **Code Coverage**. http://en.wikipedia.org/wiki/Code_coverage

[29] **Apache Log4j**. http://logging.apache.org/log4j/2.x/

[30] **JUnit**. http://junit.org/

CloudStudio API Reference

Introduction

The CloudStudio API exposes an interface to access and manipulate CloudStudio resources. All CloudStudio resources are accessed and manipulated in a similar way.

Requests to the CloudStudio API have to use either the GET or POST method. GET requests are used for functions that do not change the state of the database. POST requests are used for functions that make changes to the database.

The content type of requests to the CloudStudio API must be `application/x-www-form-urlencoded`. The response has content type `application/json`. This asynchronism allows us to provide parameters for both GET and POST requests similarly and still retrieve comprehensive JSON objects, and is used by many widely used APIs (e.g. SoundCloud).

/api/login

Method: POST

Log into CloudStudio with your username and password. Returns a session ID that will be required for further API calls, as well as the username and user privileges.

Parameters

Parameter name	Description
username	Your username
password	Your password

Example

Request

```
curl "http://cloudstudio:7330/api/login" \
  -d "username=John" \
  -d "password=burgers"
```

Response

```
{
   "sessionId": "f40309335f82e044fa04c6f267aa62fd",
   "username": "John",
   "isAdmin": false,
   "isCreator": false
}
```

/api/repositories

Method: GET

Retrieves a list of all repositories you have access to.

Parameters

Parameter name	Description
sessionId	Your session ID

Example

Request

```
curl "http://cloudstudio:7330/api/repositories?\
  sessionId=YOUR_SESSION_ID"
```

Response

```
{
   "repositories": [
      {
         "repositoryAlias": "BankAccountDemo",
         "repositoryDescription": "Dealing with banks and accounts.",
         "repositoryUrl": "https://github.com/foo/bankaccountdemo",
         "repositoryOwner": "John",
         "users": [
            "David",
            "Isabelle",
            "John"
         ],
      },
      {
         (...)
      }
   ]
}
```

/api/repositoryInformation

Method: GET

Retrieves a list of users and branches for a given repository. Also returns the timestamp of the last origin information update.

Parameters

Parameter name	Description
sessionId	Your session ID
repositoryAlias	Repository alias

Example

Request

```
curl "http://localhost:7330/api/repositoryInformation?\
  sessionId=YOUR_SESSION_ID&\
  repositoryAlias=BankAccountDemo"
```

Response

```
{
   "repositoryAlias": "BankAccountDemo",
   "repositoryDescription": "Dealing with banks and accounts.",
   "repositoryUrl": "https://github.com/foo/bankaccountdemo",
   "repositoryUsers": [
      "David",
      "Isabelle",
      "John"
   ],
   "repositoryBranches": [
      "master",
      "test_branch"
   ],
   "lastOriginUpdate": "2015-03-27 23:09:32",
   "lastOriginUpdateDiff": "1m"
}
```

/api/setRepositoryInformation

Method: POST

Updates repository information. You need to be administrator or repository owner.

Parameters

Parameter name	Description
sessionId	Your session ID
repositoryAlias	Repository alias
repositoryDescription	Description for repository
repositoryUrl	URL to remote repository

Example

Request

```
curl "http://cloudstudio:7330/api/setRepositoryInformation" \
  -d "sessionId=YOUR_SESSION_ID" \
  -d "repositoryAlias=HelloWorld" \
  -d "repositoryDescription=This is a Hello World project." \
  -d "repositoryUrl=https://github.com/foo/helloworld"
```

Response

```
{}
```

/api/branchAwareness

Method: GET

Retrieves branch level awareness information for a repository. For every branch, the active users represent the users that have this particular branch checked out currently.

For every user in branch, a relation to the origin is given. This value can be EQUAL, AHEAD, BEHIND, FORK, LOCAL_BRANCH or REMOTE_BRANCH.

Relationship with origin	Description
EQUAL	The latest branch commit is the same for the user and the origin.
AHEAD	The user has made commits and is directly ahead of the origin.
BEHIND	New commits have been pushed to the origin and the user is directly behind.
FORK	The user has made commits but other new commits have been pushed to the origin in the meantime.
LOCAL_BRANCH	This branch is a local branch for the user.
REMOTE_BRANCH	This branch only exists on the remote but not on the user's local repository.

For the relationships AHEAD, BEHIND and FORK a distance specifies the shortest distance between the current commit for the client and the origin.

For every user, the "lastUpdate" field refers to the last time that the client has sent an update to CloudStudio. "lastUpdateDiff" is the elapsed time since the last update, e.g. "2h" (2 hours) or "5d" (5 days).

Parameters

Parameter name	Description
sessionId	Your session ID
repositoryAlias	Repository alias

Example

Request

```
curl "http://localhost:7330/api/branchAwareness?\
  sessionId=YOUR_SESSION_ID&\
  repositoryAlias=BankAccountDemo"
```

Response

```
{
   "branches": [
      {
         "branch": "master",
         "activeUsers": [
            {
               "username": "David",
               "lastUpdate": "2015-03-27 20:10:03",
               "lastUpdateDiff": "3h"
            },
            {
               "username": "John",
               "lastUpdate": "2015-03-27 21:33:41",
               "lastUpdateDiff": "2h"
            }
         ],
         "users":[
            {
               "username": "David",
               "relationWithOrigin": "FORK",
               "distanceFromOrigin": 2
            },
            {
               "username": "Isabelle",
               "relationWithOrigin": "EQUAL"
            },
            {
               "username": "John",
               "relationWithOrigin": "AHEAD",
               "distanceFromOrigin": 1
            }
         ]
      },
      {
         (...)
      }
   ]
}
```

/api/fileAwareness

Method: GET

Retrieves file level awareness information for a repository and branch. All your files in a branch are compared to every other users' files in the same or specified branch.

For every user a conflict type is set to either NO_CONFLICT, FILE_CONFLICT or CONTENT_CONFLICT.

Conflict type	Description
NO_CONFLICT	The two files being compared are identical.
FILE_CONFLICT	The two files being compared are different.
CONTENT_CONFLICT	After further analysing conflicting files, by doing a three-way diff with a suitable common ancestor of both files, a merge conflict occurs.

Non-existing files are treated as empty files for this purpose.

Parameters

Parameter name	Description
sessionId	Your session ID
repositoryAlias	Repository alias
branch	Branch from which your files are compared
compareToBranch	Branch to which files of other users your files are compared to
showUncommitted	If true, also takes into account changes that have not yet been locally committed.
showConflicts	If true, for all files with a FILE_CONFLICT, additionally run a content conflict analysis. If false, just compare the files by their hash.
viewAsOrigin	Instead of showing from your perspective, show from the perspective of the origin ("true" or "false")

Example

Request

```
curl "http://cloudstudio:7330/api/fileAwareness?\
  sessionId=YOUR_SESSION_ID&\
  repositoryAlias=BankAccountDemo&\
  branch=master&\
  compareToBranch=master&\
  showUncommitted=false&\
  showConflicts=true&\
  viewAsOrigin=false"
```

Response

```
{
   "files": [
      {
         "filename": "README",
         "users": [
            {
               "username": "David",
               "type": "FILE_CONFLICT"
```

```
            },
            {
               "username": "Isabelle",
               "type": "NO_CONFLICT"
            },
            {
               "username": "John",
               "type": "NO_CONFLICT"
            }
         ]
      },
      {
         "filename": "src/java/Main.java",
         "users": [
            {
               "username": "David",
               "type": "NO_CONFLICT"
            },
            {
               "username": "Isabelle",
               "type": "CONTENT_CONFLICT"
            },
            {
               "username": "John",
               "type": "NO_CONFLICT"
            }
         ]
      }
   ]
}
```


/api/contentAwareness

Method: GET

Compares two files directly to each other.

The response contains a line-by-line comparison designed to be easily displayable in a side-by-side view.

For each line, a type is set as follows:

Type	Description
UNCHANGED	No changes have been made to this line.
INSERT	This line has been inserted.
MODIFIED	This line has been modified.
PAD	Padding for unmodified blocks to line up nicely.
MODIFIED_PAD	Padding for modified blocks to line up nicely.

Parameters

Parameter name	Description
sessionId	Your session ID
repositoryAlias	Repository alias
filename	Filename
branch	Your branch
theirUsername	User you want to compare to
compareToBranch	Branch you want to compare to
showUncommitted	If true, also take into account changes that have not yet been locally committed.
viewAsOrigin	Instead of showing from your perspective, show from the perspective of the origin ("true" or "false")

Example

Request

```
curl "http://cloudstudio:7330/api/contentAwareness?\
  sessionId=YOUR_SESSION_ID&\
  repositoryAlias=BankAccountDemo&\
  filename=README&\
  branch=master&\
  compareToBranch=master&\
  theirUsername=David\
  showUncommitted=false&\
  viewAsOrigin=false"
```

Response

```
{
   "content": [
      {
         "myContent": "Welcome to BankAccountDemo!",
         "myType": "UNCHANGED",
         "theirContent": "Welcome to BankAccountDemo!",
         "theirType": "UNCHANGED"
      },
      {
         "myContent": "",
         "myType": "PAD",
         "theirContent": "This is a project dealing with banks and accounts."
         "theirType": "INSERT"
      }
   ]
}
```

/api/contentConflict

Method: GET

Compares two files and the nearest common ancestor to each other.

The response contains a line-by-line comparison designed to be easily displayable in a side-by-side view.

For each line, a type is set as follows:

Type	Description
UNCHANGED	No changes have been made to this line.
MODIFIED	This line has been modified.
MODIFIED_PAD	Padding for modified blocks to line up nicely.
CONFLICT	This line is conflicting.
CONFLICT_PAD	Padding for conflict blocks to line up nicely.
PAD	Padding for the blocks to line up nicely.

By definition, a conflict occurs when all three lines have been modified or only the common ancestor has been modified.

Parameters

Parameter name	Description
sessionId	Your session ID
repositoryAlias	Repository alias
filename	Filename
branch	Your branch
theirUsername	User you want to compare to
compareToBranch	Branch you want to compare to
showUncommitted	If true, also take into account changes that have not been locally committed yet.
viewAsOrigin	Instead of showing from your perspective, show from the perspective of the origin ("true" or "false")

Example

Request

```
curl "http://cloudstudio:7330/api/contentAwareness?\
  sessionId=YOUR_SESSION_ID&\
  repositoryAlias=BankAccountDemo&\
  filename=src/java/Main.java&\
  branch=master&\
  compareToBranch=master&\
  theirUsername=Isabelle&\
  showUncommitted=false&\
  viewAsOrigin=false"
```

Response


```
{
   "content":[
      {
         "myType": "UNCHANGED",
         "myContent": "First line.",
         "theirType": "UNCHANGED",
         "theirContent": "First line.",
         "baseType": "UNCHANGED",
         "baseContent": "First line."
      },
      {
         "myType": "MODIFIED",
         "myContent": "Only I changed this, no worries.",
         "theirType": "MODIFIED",
         "theirContent": "Second line.",
         "baseType": "MODIFIED",
         "baseContent": "Second line."
      },
      {
         "myType": "CONFLICT",
         "myContent": "I made a change.",
         "theirType": "CONFLICT",
         "theirContent": "Third line.",
         "baseType": "CONFLICT",
         "baseContent": "Me too! Whoops."
      }
   ]
}
```


/api/users

Method: GET

Retrieves a list of all users, their privileges and the date they created the account. Must be administrator to perform this operation.

Parameters

Parameter name	Description
sessionId	Your session ID

Example

Request

```
curl "http://cloudstudio:7330/api/users?\
  sessionId=YOUR_SESSION_ID"
```

Response


```
{
   "users": [
      {
         "username": "Admin",
         "joinDate": "2015-02-01 16:23:12",
```

```
            "isAdmin": true,
            "isCreator": true,
        },
        {
            "username": "David",
            "joinDate": "2015-03-02 23:01:57",
            "isAdmin": false,
            "isCreator": true,
        },
        {
            "username": "Isabelle",
            "joinDate": "2015-03-07 11:23:01",
            "isAdmin": false,
            "isCreator": false,
        },
        {
            "username": "John",
            "joinDate": "2015-03-16 10:13:41",
            "isAdmin": false,
            "isCreator": true,
        },
    ]
}
```


/api/createRepository

Method: POST

Creates a new repository and sets its owner to yourself. Repository creation rights are required for this operation.

Parameters

Parameter name	Description
sessionId	Your session ID
repositoryAlias	Repository alias
repositoryUrl	URL to the remote, e.g. GitHub
repositoryDescription	A short description

Example

Request

```
curl "http://cloudstudio:7330/api/createRepository" \
  -d "sessionId=YOUR_SESSION_ID" \
  -d "repositoryAlias=HelloWorld" \
  -d "repositoryDescription=This is a Hello World project." \
  -d "repositoryUrl=https://github.com/foo/helloworld"
```

Response


```
{}
```

/api/deleteRepository

Method: POST

Deletes a repository. You need to be administrator or the repository owner for this operation.

Parameters

Parameter name	Description
sessionId	Your session ID
repositoryAlias	Repository alias

Example

Request

```
curl "http://cloudstudio:7330/api/deleteRepository" \
  -d "sessionId=YOUR_SESSION_ID" \
  -d "repositoryAlias=HelloWorld"
```

Response

```
{}
```

/api/addUserToRepository

Method: POST

Adds a user to a repository. Must be repository owner or administrator.

Parameters

Parameter name	Description
sessionId	Your session ID
repositoryAlias	Repository alias
username	Username

Example

Request

```
curl "http://cloudstudio:7330/api/addUserToRepository" \
  -d "sessionId=YOUR_SESSION_ID" \
  -d "repositoryAlias=HelloWorld" \
```

```
  -d "username=David"
```

Response

```
{}
```

/api/removeUserFromRepository

Method: POST

Removes a user from a repository. Must be repository owner or administrator.

Parameters

Parameter name	Description
sessionId	Your session ID
repositoryAlias	Repository alias
username	Username

Example

Request

```
curl "http://cloudstudio:7330/api/removeUserFromRepository" \
  -d "sessionId=YOUR_SESSION_ID" \
  -d "repositoryAlias=HelloWorld" \
  -d "username=David"
```

Response

```
{}
```

/api/modifyRepositoryOwner

Method: POST

Sets a new repository owner. Must be repository owner or administrator.

Parameters

Parameter name	Description
sessionId	Your session ID
repositoryAlias	Repository alias
username	New repository owner

Example

Request

```
curl "http://cloudstudio:7330/api/modifyRepositoryOwner" \
  -d "sessionId=YOUR_SESSION_ID" \
  -d "repositoryAlias=HelloWorld" \
  -d "username=David"
```

Response

```
{}
```

/api/createUser

Method: POST

Creates a new user.

Parameters

Parameter name	Description
username	Username
password	New password

Example

Request

```
curl "http://cloudstudio:7330/api/createUser" \
  -d "username=David" \
  -d "password=penguins"
```

Response

```
{}
```

/api/deleteUser

Method: POST

Removes a user. Requires administrator privileges.

Parameters

Parameter name	Description
sessionId	Your session ID
repositoryAlias	Repository alias
username	Username

Example

Request

```
curl "http://cloudstudio:7330/api/deleteUser" \
  -d "sessionId=YOUR_SESSION_ID" \
  -d "username=David"
```

Response

```
{}
```

/api/changePassword

Method: POST

Changes a user's password.

Parameters

Parameter name	Description
sessionId	Your session ID
newPassword	New password

Example

Request

```
curl "http://cloudstudio:7330/api/changePassword" \
  -d "sessionId=YOUR_SESSION_ID" \
  -d "newPassword=polarbears"
```

Response

```
{}
```

/api/giveAdminPrivileges

Method: POST

Give administrator privileges to a user. Requires administrator privileges.

Parameters

Parameter name	Description
sessionId	Your session ID
repositoryAlias	Repository alias
username	Username

Example

Request

```
curl "http://cloudstudio:7330/api/giveAdminPrivileges" \
  -d "sessionId=YOUR_SESSION_ID" \
  -d "username=David"
```

Response

```
{}
```

/api/revokeAdminPrivileges

Method: POST

Revoke a user's administrator privileges. Requires administrator privileges.

Parameters

Parameter name	Description
sessionId	Your session ID
repositoryAlias	Repository alias
username	Username

Example

Request

```
curl "http://cloudstudio:7330/api/revokeAdminPrivileges" \
  -d "sessionId=YOUR_SESSION_ID" \
```

```
  -d "username=David"
```

Response

```
{}
```

/api/giveCreatorPrivileges

Method: POST

Give repository creation privileges to a user. Requires administrator privileges.

Parameters

Parameter name	Description
sessionId	Your session ID
repositoryAlias	Repository alias
username	Username

Example

Request

```
curl "http://cloudstudio:7330/api/giveCreatorPrivileges" \
  -d "sessionId=YOUR_SESSION_ID" \
  -d "username=David"
```

Response

```
{}
```

/api/revokeCreatorPrivileges

Method: POST

Revoke a user's repository creation privileges. Requires administrator privileges.

Parameters

Parameter name	Description
sessionId	Your session ID
repositoryAlias	Repository alias
username	Username

Example

Request

```
curl "http://cloudstudio:7330/api/revokeCreatorPrivileges" \
  -d "sessionId=YOUR_SESSION_ID" \
  -d "username=David"
```

Response

```
{}
```

/api/localState

Method: POST

Update the server with the user's git repository information for a single repository. This operation is used by the client periodically.

For this operation only, repository data needs to be sent as `application/json`.

Parameters

Parameter name	Description
sessionId	Your session ID
repositoryAlias	Repository alias

Example

Request

```
curl "http://cloudstudio:7330/api/localState?sessionId=YOUR_SESSION_ID&reposi
  -H "Content-Type: application/json" \
  -d "$JSON_STRING"
```

With the `$JSON_STRING` being:

```
{
   "files": [
      {
         "filename": "README",
         "branch": "master",
         "content": "This is the read-me file.",
         "committed": "committed",
         "commit": "65fcfcd7860bf95fd1dce7c01bcd886bcdf4e675"
      },
      {
         "filename": "README",
         "branch": "master",
         "content": "I made some uncommitted changed to the read-me file.",
         "committed": "uncommitted",
         "commit": "65fcfcd7860bf95fd1dce7c01bcd886bcdf4e675"
```

```
        }
    ],
    "branches": [
        {
            "commit": "73e68dd8ae12bdf7dfce4a29cd0a6cb6ce99aca8",
            "active": true,
            "branch": "master"
        }
    ],
    "commitHistory": [
        {
            "commit": "73e68dd8ae12bdf7dfce4a29cd0a6cb6ce99aca8",
            "downstreamCommits":[
                {
                    "distance": 0,
                    "commit": "73e68dd8ae12bdf7dfce4a29cd0a6cb6ce99aca8"
                },
                {
                    "distance": 1,
                    "commit": "fdaa47da4ab7f22fc06373c407c48326e70db199"
                }
            ]
        }
    ]
}
```


Response


```
{}
```


Error handling

An erroneous request results in a status code 400 (Bad Request) response. The response data is a JSON object as always and contains and error message.

Example

Response


```
{
    "error": "Insufficient privileges"
}
```