INTEGRATING VERSION CONTROL INTO OWNCLOUD

Progress Report

STUDENT: Craig Roberts (cgr9@aber.ac.uk)
SUPERVISOR: Richard Shipman (rcs@aber.ac.uk)

Open Source Computing (G402)
Department of Computer Science
Aberystwyth University

Version 1.1 (Release)

4,394 words

CONTENTS

Part I

AN INTRODUCTION

1

PROJECT SUMMARY

ownCloud is a FLOSS project aiming to deliver a self-hosted cloud solution similar to Dropbox and Ubuntu One. It is being developed under the umbrella of the KDE project, started by Frank Karlitschek at Camp KDE'10 [1].

Currently ownCloud offers several features: file storage with WebDAV access (supported by all three major operating systems), contacts and calendar syncing with CalDAV, a PHP-only implementation, and the sharing of file or folder access between users.

This project aims to integrate a version control system into the filesystem abstraction library used by ownCloud. Practically, this involves implementing a filesystem reader for ownCloud that can read a version control repository. To gain a better understanding of the mechanisms used by version control systems, a review of existing literature and implementations has been carried out.

According to the Undergraduate Information page for the Open Source Computing degree:

> The final year project is expected to be a major project developed using open source principles.

ownCloud is licensed under the AGPLv3 (Affero GNU Public Licence Version 3) which is classified as an open-source licence. It is developed using open source principles, with development taking place in a public Git repository hosted by Gitorious [1].

Finally, version control systems vary widely in their implementations, with many of their strengths and capabilities arising from the choice of algorithms and data structures. This project provides practical learning opportunities for how the major version control systems work, providing a clearer understanding of which is more suitable for different situations. Successfully integrating version control into ownCloud would also make it one of the first fully open-source cloud solutions with features comparable to Dropbox and Ubuntu One.

1 Public git repository hosted at http://gitorious.org/owncloud/owncloud

2

BACKGROUND

There has been some prior discussion on the mailing list about using Git as a method of implementing version control into ownCloud. The arguments for Git are its simplicity, and the existence of PHP libraries. However, there do not appear to be any public implementations.

Several projects, mostly on the GitHub hosting service, provide various levels of functionality for interfacing with Git from PHP 5. A thorough analysis of these projects is currently in progress as part of the Version Control Comparison Report. The report will be provided as a deliverable along with the final report.

There are also several related projects using version control systems for backup. Many distributed version control systems have issues with large files, which has led to one or two projects per system aiming to provide patches.

2.1 LARGE FILE ISSUES

Distributed version control systems have issues with large binary files; the effect on centralised systems is reduced, as most systems store the history on the server and check out only the most recent copies.

Binary files cannot be diffed very efficiently; compressed binary formats are even worse, because changes early on in the byte-stream cascade throughout, causing larger deltas. When given two binary files as input, `git diff` reports only that the two are different [5]. Mercurial operates in a similar manner [3].

Since the diff algorithm is reporting only the files are different, a DVCS will store an entirely new snapshot in the repository. For large binary files, this causes the repository to grow very quickly, especially the more often the binary files change. This bottleneck has led to the Bup project, discussed below, which attempts to use an rsync-like algorithm along with "chunking", before using `diff`. This allows Bup to efficiently version large files, both in terms of disk storage and memory usage.

There is also the git-annex project. git-annex moves large files out of the repository, into a private store inside the `.git` folder [2]. Symbolic links are then created and checked into the repository instead. This allows the tracking of large files within a repository, without storing the files themselves. The trade-off is no version history is kept for the large files either.

Finally, Git has a fix for the out-of-memory error that occurs with files larger than the available system memory in version 1.7.6. The `core.bigfilethreshold` configuration value tells Git to write large files directly to packfiles rather than attempting to load them into memory.

It appears that no distributed version control system is suitable for handling large binary files, although work is progressing on solutions. For simplicity, this project will not focus on supporting large files, although Bup may provide a solution for Git-compatibility.

2.2 BUP—LARGE FILE BACKUP

Bup is a backup system based on the Git packfile format. Bup is an interesting implementation for several reasons; most of this information was taken from the Bup design document in the public repository [1].

Bup uses an rsync-like rolling checksum algorithm [10] to split large files into chunks along adler32 boundaries, which are then versioned independently. Using the adler32 checksum algorithm for chunking provides an advantage for compressed binary files: inserting/deleting bytes changes, at most, 2*(log n) chunks.

Just like Git, Bup writes these chunks into packfiles, which provides Git compatibility. The caveat is that Git will only be able to see the myriad of chunks Bup has created, rather than the original files.

Since these chunks are all individual Git blobs, the normal Git rules apply to them, which include Git's efficient storage mechanism. Any modification in a large file only changes a few chunks. Therefore, between versions, the rest of the file isn't modified, meaning the chunks stored previously can be re-used. This leads to extremely efficient de-duplication when storing large files.

The Bup documentation also discusses a limitation of this approach. Packfiles are extremely efficient for storing data, but to provide quicker access to the objects stored inside them, Git uses idx files with "bookmarks" into the file [7]. This breaks down with the large number of chunks that Bup commits to the repository.

The solution Bup has favoured is another layer of abstraction: midx files, which index the idx files. The midx files are a bup-specific

1 The Bup design document was taken from: https://github.com/apenwarr/bup/blob/master/DESIGN

optimisation, but since they are stored separately Git can still read the repository.

It may be possible to write code to read Bup index files, providing an interface to Bup repositories as well. Since they both store data in the packfile format, reading and writing will be broadly similar for both systems. This is too much work for this project, but is mentioned as a future possibility.

2.3 GLIP–A PHP GIT LIBRARY

Glip is a Git library written entirely in PHP5 [8]. It is capable of reading and writing to Git packfiles, allowing PHP access to Git repositories. Glip writes directly to packfiles, allowing more complex functionality to be implemented as required.

Glip cannot initialise a Git repository, however, and does not provide push/pull support. It is possible to use the functionality provided to create an initial commit and to attempt to commit it to the repository, but there are currently implementation issues leading to a corrupt repository.

2.4 SPARKLESHARE–GIT-BASED DESKTOP SYNC

Sparkleshare is a collaboration and sharing tool, using Git repositories as the backend storage solution. There are clients for Linux, Mac OS X and Android [6]. Sparkleshare offers users the option to host their own server, using a system with Git and an SSH server installed.

The desktop client allows users to choose which folders to sync to their Sparkleshare folder. The desktop client synchronises the folder from the remote source using Git and monitors the folders for changes.

Support already exists for self-hosted Git repositories, GitHub and Gitorious. Providing a Git implementation for ownCloud would allow support for using an ownCloud server automatically. This would also require the implementation of push and pull support for the Git implementation, through one of Git's transport mechanisms. This would enable the Sparkleshare client to act as a desktop client to ownCloud, without any modification to the Sparkleshare source code.

2.5 PHP IMPLEMENTATIONS

While Glip provides read and write functionality for Git, there are other projects which claim to provide push and pull support over PHP. The VCS Comparison Report explores this area in detail, and is still a work-in-progress, but it has clarified the reasons for choosing Git

over other distributed systems. Three systems were considered: Git, Mercurial, and Bazaar.

Bazaar is a fascinating piece of software, capable of offering multiple workflows based on either centralised, decentralised or 'hybrid' models. As such it would take longer than the time available to fully investigate Bazaar as a solution.

Mercurial and Git are each relatively simple, in both operation and repository format. The decision to use Git implies that Mercurial could also be used very easily; however, Git has the advantage of multiple PHP libraries, a *very* simple repository format and the Sparkleshare front-end.

3

GOALS AND OBJECTIVES

3.1 READING AND WRITING GIT REPOSITORIES

Using Glip it should be possible to build a working implementation to read existing Git repositories. This would require reading commit objects from the repository and then accessing the tree associated with it. The tree object provides references to individual blobs, which can be read directly from the packfiles using Glip. Mimetype checking will be required in order to determine what file format is being manipulated.

3.2 GIT PHP LIBRARY

Building on the support Glip has for reading and writing packfiles, a PHP library with support for more easily committing several files, accessing previous revisions of files, and tracking and storing metadata, will be required.

The Git PHP library will provide support for reading and writing directly to Git packfiles, index files and loose objects. There will also be a higher-level interface using this functionality, providing commands analogous to the Git commands (commit, status, checkout, etc.).

3.3 GIT INIT SUPPORT

In order to create Git repositories upon installation, the library must be able to support initialising a Git repository. This involves creating a bare directory structure and initialising some simple text files. Provided the repository is initialised correctly, the initial commit process should be no different to any subsequent commits.

3.4 OWNCLOUD FILESYSTEM CLASS IMPLEMENTATION

ownCloud already uses an abstraction layer for the filesystem access. By creating an empty folder in the user's data folder and providing an

`OC_Filestorage` class implementation, ownCloud passes all the read and write operations through the directory handle provided by the `OC_Filestorage::opendir()` method.

The `OC_Filestorage` class can be used to abstract the Git repository from the ownCloud filesystem. Using a simple configuration setting, the class can load any previous commit and read the state of the files. This is returned to ownCloud, which continues to process the files as normal, allowing download, web viewing, and WebDAV access.

3.4.1 *Initial Acceptance Tests*

The following subsections describe the user-level functionality that the ownCloud implementation must satisfy. This specifies a minimum set of requirements to be delivered at the end of the project.

3.4.1.1 *Filesystem Viewing*

The system must be able to view a Git repository at any point in time. Specifically, the user must be able to see the filename, the object type (file or directory), the file size and the last modified timestamp.

3.4.1.2 *Metadata Viewing*

The user must be able to view the file size and the last modified timestamp of the file. If there is no metadata file available for the directory or file, the last modified timestamp is assumed to be the same as the last commit timestamp.

If a metadata file is present, it is parsed and the correct last modified timestamp is displayed instead.

3.4.1.3 *File Download*

The system must be able to provide normal viewing capabilities for files stored in Git. For example, text files may be displayed in the browser if the Text Editor extension is enabled. Non-text files are passed directly to the browser, which downloads them to the user's hard drive.

The filesystem viewer will provide access to the correct revision, when the user selects a file it must perform the same action as a local file would.

3.4.1.4 *Deletion and Recovery*

The system must be able to remove files from a Git repository, and recover any file from a previous version by creating a new commit. The user must be able to select a previous version of any file, or select a deleted file, and perform a checkout. The file must be restored to its original location.

3.4.1.5 *Push and Pull Support*

It must be possible to push changes to the Git repository over the smart HTTP protocol. The system must also provide support for the dumb HTTP protocol. WebDAV push is configured using Apache and WebDAV, and does not need to be implemented as part of this project. Pushing must not cause any conflicts, errors, or leave the server-side repository in an inconsistent state. The system must support pushing to any branch, or the creation of a new branch where necessary.

The system must also support dumb and smart HTTP pulling. Conflict resolution is not required here, as the client takes care of it instead. The system must allow pulling of any branch, not necessarily the `master` branch.

3.5 GIT PHP LIBRARY

The Git PHP library will consist of a set of classes that represent the objects in the Git repository. An extra class will provide commands similar to those provided by the Git client, including `add`, `commit`, `checkout`, etc. A more detailed explanation can be found in section 5

3.6 METADATA SUPPORT

Git does not track metadata, apart from the executable bit; a mechanism will be needed to provide metadata to the ownCloud interface, and also to the WebDAV clients. The current ownCloud implementation uses the PHP `stat()` function, which returns metadata such as the owner uid, the group guid, access and modification timestamps, and the size in bytes.

It is possible to access metadata for loose objects directly, but for packfiles only the uncompressed object size is stored in the header. The most appropriate method, which is also used by the Bup project, would be to track file metadata in a dot-prefixed file for each directory tree. This file could be a lightweight JSON file or similar, and be added to the repository for versioning. This file would then contain metadata for each version of each file.

The metadata could be stored server-side, in a database or similar cache, which updates every time a commit or push is executed. However, this restricts the metadata information to the server, while it might be useful to have it synchronise to client repositories as well. The exact file format can be modified later, since the values stored are not likely to change.

Part II

CURRENT PROGRESS

4

TECHNICAL CHALLENGES

4.1 READING AND WRITING GIT REPOSITORIES

A version control repository is often implemented as a series of files, stored in some common database such as a `.svn` or `.git` folder. The formatting of these files differs between systems, depending on the data structures inherent in the system design. Git uses the packfile format and an associated index file for low-level storage; a detailed overview is available in Daniel Kuhn's Master's Thesis [9].

4.2 PUSH AND PULL SUPPORT

In order to provide full compatibility with existing Git clients, it must be possible to pull from the ownCloud Git repository, and push changes to it from a desktop client. Git supports "smart" pushing over HTTP as of 1.6.6 [4].

Prior to 1.6.6, "dumb" protocols are supported. This involves using HTTP to fetch loose objects and packfiles with standard HTTP GET requests. This is inefficient, as Git has to request the entire packfile and then find the object within it. Similarly, when pushing over HTTP, a WebDAV server is required (which ownCloud supports) but this is also inefficient compared to the other "smart" protocols [4]. The advantage is that only a standard HTTP web server is required, with WebDAV optional for push support.

As of version 1.6.6, Git has support for "smart" HTTP, which requires Git to be installed on the server. Git uses the same GET requests as the dumb protocol described above, but with additional GET parameters, specifying a command to run, a path, and a hash of the object being requested. This allows Git to requests specific objects which are unpacked by the server instead of transferring the entire packfile to the client. Current implementations use the Git binary: the core git distribution includes a CGI script to handle the HTTP requests.

4.3 CONFLICT RESOLUTION

One of the most difficult aspects of implementing distributed version control is conflict resolution. One possibility is a "win" model where one commit wins based on some attribute, such as the time of the commit. The alternative is the behaviour of Ubuntu One and Dropbox, where the conflicting copy is labelled `$FILENAME.conflict.$NUM` or similar.

The ownCloud mailing list will be able to provide advice on the preferred option; the ideal solution would require no user interaction.

5

OUTLINE DESIGN

5.1 OPERATING ENVIRONMENT

ownCloud is expected to run on the Apache web server, along with an SQL database server and write permissions to two directories: `/data/` and `/config/`. ownCloud displays the files under the `/data/$USERNAME/files` directory through WebDAV and the file explorer, leaving the top-level `/data/$USERNAME/` folder available for repository storage with write permissions.

5.2 DESIGN AND IMPLEMENTATION CONSTRAINTS

In order not to create any new dependencies, the filesystem modification for repository reading must not require the version control system to be installed on the server. The solution is to provide a PHP library which abstracts reading and writing to the repository. An `OC_Filestorage` class implementation will then provide access to this library, presenting an ordinary filesystem view to the rest of ownCloud.

5.3 ASSUMPTIONS AND DEPENDENCIES

The ownCloud software requires PHP 5.2 or greater, and an SQL database (MySQL and SQLite are officially supported). Apache is the officially supported web server, although advanced users should have no trouble configuring an alternative. No other dependencies are required, or desirable.

There is an assumption that the version control system is not installed on the server. While this will be true in most cases, there may also be cases where the it is available as an installed binary. In this case, the library should utilise the system binary where it makes sense for performance/optimisation reasons.

5.4 OWNCLOUD IMPLEMENTATION

ownCloud is modelled on the Model–View–Controller design pattern. The initial request to the application goes to the `index.php` file, which implements a basic event dispatcher. The implementation will involve writing an `OC_Filestorage` class which uses the Git PHP library to access the repository. A tentative class diagram can be found in Appendix B.

A simplified request to `index.php` looks something like:

- includes `lib/base.php` which provides a standard `OC` (ownCloud) class with some useful constants
- executes `OC::init()`, which performs basic set-up and registers the `OC_Filestorage` classes as providers – the Git Filestorage provider will hook in here
- Execution is handed back to `index.php` which checks authentication
- If the user is logged in, they are redirected `files/index.php` which provides an AJAX interface to the filesystem

6

IMPLEMENTATION OPTIONS AND CHOICES

There are very few implementation decisions to be made for this project. ownCloud is written in PHP, and requires no additional dependencies aside from an SQL database—maintaining this is one of the key requirements of contributions.

The version control system decided upon could well have been Mercurial, or even another DVCS. Subversion support was briefly considered due to the use of WebDAV in the specification, however, this would require the implementation of the Delta-V specification for SabreDAV [1]. In any case, there is much more example code for interacting with Git repositories than its competitors, at least for PHP.

Compared to traditional Git clients, branching and merging is not expected to happen often, except between cloned repositories. In this case conflicts result in two files, rather than requiring resolution. This avoids having to perform diffs and merges in PHP. The user can simply open both files, decide which version to keep and continue with their work. User testing during development will provide more feedback on this aspect.

ownCloud itself is an excellent choice of project; there exist several open-source projects that attempt to use Git for synchronisation and backup, but ownCloud is one of the simplest to install, set up and use. It is also supported and maintained by prominent members of the KDE community.

PHP is a logical implementation language, although it will never be as efficient as the Git client, written in Perl and C. PHP has widespread support amongst web hosting providers, and makes it very simple to implement the HTTP communication with Git.

The most complex algorithms to produce in PHP will be writing and extracting objects to and from packfiles. The binary format for packfiles has two versions, although they are largely similar. Push and pull support, which implements synchronisation, can be achieved with the conflict resolution described earlier and the reading and writing implemented by the PHP Git library—no complex algorithms are necessary.

1 SabreDAV is the PHP library used by ownCloud to implement WebDAV access

Part III

PROJECT PLANNING

7

PROCESS MODEL

As a solo development project, team-based methodologies such as Agile or XP are not entirely suitable. Equally, the traditional Waterfall model is not particularly suited to the open-source nature of ownCloud—the development branch is updated regularly, and the project currently uses a three-month release cycle. ownCloud 3 is due for release in January 2012, with ownCloud 4 scheduled for the end of April. If this project is accepted into the master branch, ownCloud will have released two major versions by the time it is complete.

Iterative development allows for successive iterations to be released alongside the ownCloud development branch, until a final release can be made once the ownCloud APIs are stable. Used in conjunction with test-driven development, it can be verified that the software is always functioning correctly with the latest development version. This permits more confidence when developing new features and reduces the risk of external changes to ownCloud breaking the build. Finally, code coverage tools will ensure proper test coverage for the code base.

In terms of this project, an iteration results in a milestone, which is linked to a deliverable. Seven milestones are displayed on the project Gantt chart in Appendix A, including the progress report and the final report. Slack has been provided in tasks 2.6 and 2.8 which can be utilised if previous tasks take longer than expected.

8

WEEKLY PLAN

A Gantt chart detailing tasks, durations, start and end dates is available in Appendix A. The tasks displayed below are in roughly chronological order and correspond to the project Gantt chart.

1. **Prototyping (21 Nov.–30 Jan.)**

   1.1. **Reading and Writing Git Repositories:** A prototype involving reading and writing commits and objects to local Git repositories will be completed by Monday 28th.

   21 Nov.–27 Nov.

   1.2. **Demonstration:** Demonstrate current progress and a working implementation to the project supervisor at the next meeting, Wednesday 30th November. The demonstration will involve viewing files, metadata and previous revisions.

   30 Nov.

   1.3. **Alpha Prototype:** An alpha prototype by early-to-mid December will provide the basic implementation structure, with stub classes providing an initial idea of the API. An ownCloud implementation must be working, and it must provide a reading and writing to the repository via ownCloud.

   7 Dec.

   1.4. **Investigate Git Repository Structure:** Investigate the directory structure of a Git repository as part of the Version Control Comparison report, which will then be used to initialise Git repositories through ownCloud. It is also necessary to investigate how the three major data structures (commits, blobs and trees) are linked together, particularly in which order they are generated. The current implementation is corrupting the repository when attempting to write new objects.

   21 Nov.–23 Nov.

   1.5. **Initialising Git Repositories:** Develop a prototype to initialise and commit to Git repositories that have not been

initialised by the Git client. The PHP-only requirement is directly dependent on this functionality.

5 Dec.–18 Dec.

1.6. **Beta Prototype:** Building on the alpha prototype with read and write support, release a beta prototype by the end of December. The beta prototype must support the initialisation of Git repositories without a native Git client.

25 Dec.

1.7. **Investigate Git Transport Protocols:** Git can use HTTP(S), SSH, the Git protocol, or the local filesystem to perform push and pull operations. In the case of HTTP, there are two distinct versions, a dumb protocol and a smart protocol. Both must be supported in the final implementation. Investigation will involve documenting the GET requests that the Git client makes to the HTTP server.

21 Dec.–3 Jan.

1.8. **Prototype Git Push/Pull Support:** Develop a prototype to allowing pushing to Git repositories hosted by ownCloud, as well as via the WebDAV implementation. Pulling must also be supported via HTTP. Compliance can be tested by using the Git client to interact with the ownCloud-hosted repository.

3 Jan.–23 Jan.

1.9. **Release Candidate:** With reading and writing, initialisation and push/pull support working successfully (confirmed with unit tests), a release candidate will be made available at the end of January 2012. This marks the beginning of development on the final implementation, along with a comprehensive test suite.

30 Jan.

2. **Development (19 Dec.–18 Mar.)**

2.1. **Write Unit Tests for PHP Git Library:** Building on the release candidate's functionality, the API of the library will be fairly stable. A comprehensive test suite tying everything together will be created. Considerations will include cross-platform support, functional testing for the user interface, and stress-testing the push and pull support.

19 Dec.–25 Dec.

2.2. **PHP Git Library:** The release candidate will be re-factored, improved and polished to conform to the unit tests. Documentation will also be started at this phase, including PHPDoc (a Javadoc adaptation) and a user guide. User

feedback from the release candidate will also be considered during re-factoring.

26 Dec.–15 Jan.

2.3. **Second Release Candidate:** The second release candidate will effectively mark the completion of the Git library. The remaining tasks will focus on ownCloud integration through an `OC_Filestorage` implementation.

22 Jan.

2.4. **Write Unit Tests for ownCloud Integration:** Test suites covering the filesystem implementation, WebDAV operation and regression tests for related classes will be created. The ownCloud project does not appear to have a preferred testing methodology.

16 Jan.–22 Jan.

2.5. **Implement ownCloud Integration:** Develop a Filestorage implementation that utilises the PHP Git library to provide access to the repository. User interface modifications must also be developed to provide configuration options for the Git repositories.

23 Jan.–12 Feb.

2.6. **Testing and Re-factoring:** A final round of re-factoring and testing to ensure no unexpected surprises or bugs. This phase also provides slack for the previous tasks.

13 Feb.–4 Mar.

2.7. **Alpha Release:** An initial alpha release of the ownCloud implementation, distributed along with the PHP Git library, effectively marks the end of development. The developed implementation should be fully-functional, rigorously tested and well-documented.

5 Mar.

2.8. **Community Release and Feedback:** At this point, a request will be made to the ownCloud community to consider these changes for the next release. Community testing and feedback will provide input for further development.

5 Mar.–18 Mar.

9

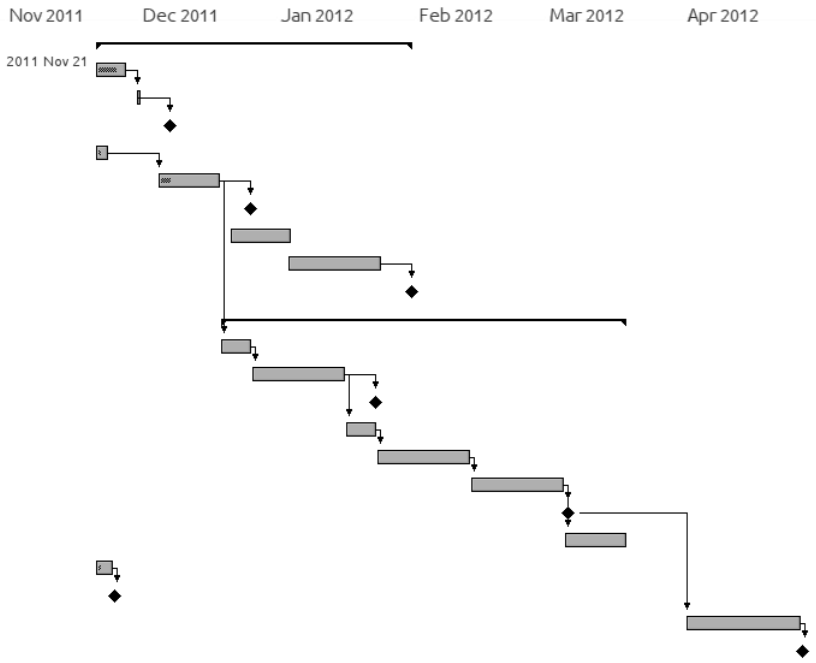
DEMONSTRATION PLAN

There are no specific demonstration requirements for this project. A laptop running a local Apache server with ownCloud installed can present the user interface changes, alternatively ownCloud can be installed on a shared hosting service, since it requires only PHP5 and an SQL database. A demonstration would involve stepping through revisions, downloading files and viewing the filesystem through the ownCloud browser or a native WebDAV client.

A

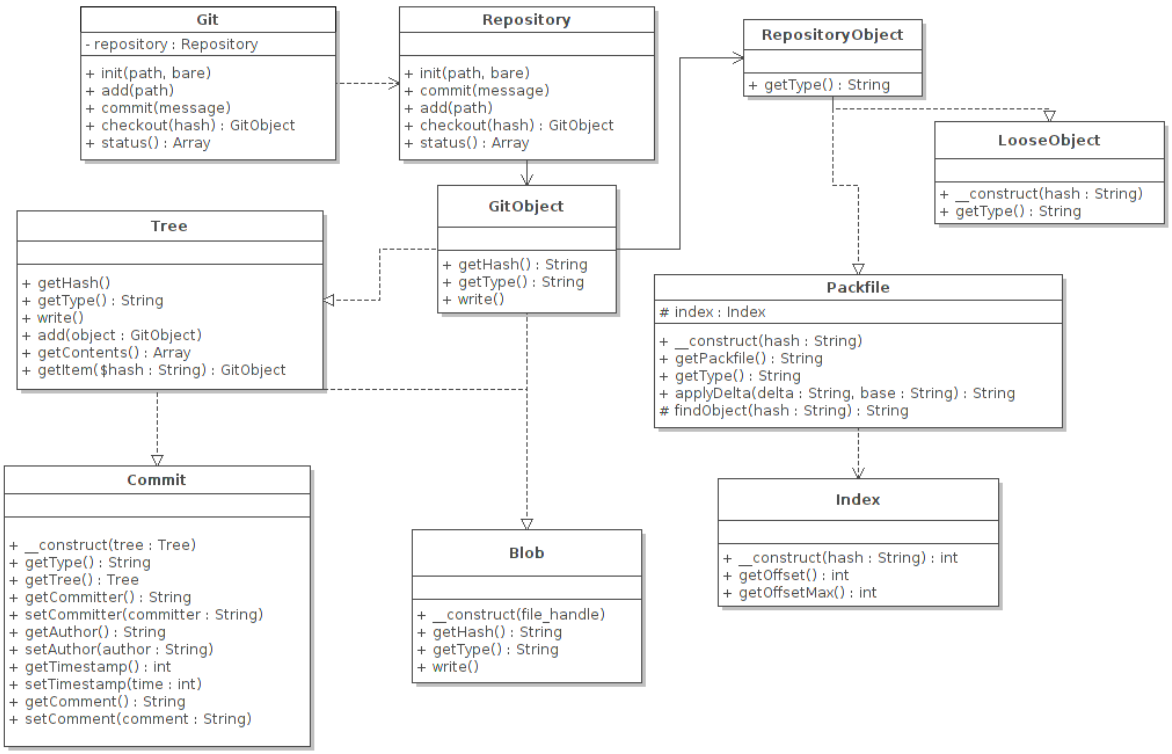
APPENDIX A

	Name	Start	Finish	Duration
1	**Prototyping**	**21 Nov**	**30 Jan**	**71d**
1.1	Reading and Writing Git Repositories	21 Nov	27 Nov	7d
1.2	Demonstration	30 Nov	30 Nov	1d
1.3	Alpha Prototype	7 Dec	7 Dec	N/A
1.4	Investigate Git Repository Structure	21 Nov	23 Nov	3d
1.5	Initialising Git Repositories	5 Dec	18 Dec	14d
1.6	Beta Prototype	25 Dec	25 Dec	N/A
1.7	Investigate Git Transport Protocols (H	21 Dec	3 Jan	14d
1.8	Prototype Git Push/Pull Support	3 Jan	23 Jan	21d
1.9	Release Candidate	30 Jan	30 Jan	N/A
2	**Development**	**19 Dec**	**18 Mar**	**91d**
2.1	Write Unit Tests for PHP Git Library	19 Dec	25 Dec	7d
2.2	PHP Git Library (Read, Write and Init)	26 Dec	15 Jan	21d
2.3	Second Release Candidate	22 Jan	22 Jan	N/A
2.4	Write Unit Tests for ownCloud Integr	16 Jan	22 Jan	7d
2.5	Implement ownCloud Integration	23 Jan	12 Feb	21d
2.6	Testing and Refactoring	13 Feb	4 Mar	21d
2.7	Alpha Release	5 Mar	5 Mar	N/A
2.8	Community Release and Feedback	5 Mar	18 Mar	14d
3	Write Progress Report	21 Nov	24 Nov	4d
4	Progress Report Hand-In	25 Nov	25 Nov	N/A
5	Write Final Report	1 Apr	26 Apr	26d
6	Final Report Hand-In	27 Apr	27 Apr	N/A



B

APPENDIX B


Git
- repository : Repository
+ init(path, bare)
+ add(path)
+ commit(message)
+ checkout(hash) : GitObject
+ status() : Array
Repository
+ init(path, bare)
+ commit(message)
+ add(path)
+ checkout(hash) : GitObject
+ status() : Array
RepositoryObject
+ getType() : String
LooseObject
+ __construct(hash : String)
+ getType() : String
GitObject
+ getHash() : String
+ getType() : String
+ write()
Tree
+ getHash()
+ getType() : String
+ write()
+ add(object : GitObject)
+ getContents() : Array
+ getItem($hash : String) : GitObject
Packfile
index : Index
+ __construct(hash : String)
+ getPackfile() : String
+ getType() : String
+ applyDelta(delta : String, base : String) : String
# findObject(hash : String) : String
Commit
+ __construct(tree : Tree)
+ getType() : String
+ getTree() : Tree
+ getCommitter() : String
+ setCommitter(committer : String)
+ getAuthor() : String
+ setAuthor(author : String)
+ getTimestamp() : int
+ setTimestamp(time : int)
+ getComment() : String
+ setComment(comment : String)
Blob
+ __construct(file_handle)
+ getHash() : String
+ getType() : String
+ write()
Index
+ __construct(hash : String) : int
+ getOffset() : int
+ getOffsetMax() : int

ANNOTATED BIBLIOGRAPHY

[1] "Discover owncloud," (http://owncloud.org/discover/), last accessed Nov. 2011.

Describes how ownCloud began with a keynote by Frank Karlitschek at Camp KDE'10, in which he discussed the need for a self-hosted, free and open-source cloud software solution.

[2] "git-annex," (http://git-annex.branchable.com/), last accessed Nov. 2011.

The git-annex project solves the large file issue in Git by taking large files *out* of the repository, and replacing them with symbolic links. These symbolic links can then be versioned, renamed and moved as normal, but without the disadvantages of dealing with large binary files. Since git-annex does not perform any modifications to Git, or to its repository, future work may allow support for git-annex repositories as well as Bup repositories.

[3] "Mercurial wiki – binary files," (http://mercurial.selenic.com/wiki/BinaryFiles), last accessed Nov. 2011.

Describes how Mercurial handles binary files for versioning. Mercurial considers files with `NULL` bytes to be binary, and will not attempt to process them. Mercurial also offers the `-a` option to some commands to force processing binary files as if they were text files.

[4] "Pro git – smart http transport," (http://progit.org/2010/03/04/smart-http.html), last accessed Nov. 2011.

Describes the difference between "smart" and "dumb" transport protocols. In the case of HTTP, it describes how "smart" implementations are backwards-compatible and how it is dealt with on the Apache side, through a CGI script. The HTTP push mechanisms are important to this project, because the alternative is pushing over WebDAV (more inefficient), SSH (which is an additional complication) or the Git protocol (for which there exists little documentation).

[5] "Pro git 7.2 – git attributes," (http://progit.org/book/ch7-2.html), last accessed Nov. 2011.

Describes how Git can be modified through configuration options to treat certain binary files as text files. This differs in Git, when compared to Mercurial, because Git provides filters for the binary data through configuration settings. These filters can process binary files and produce human-readable output from them, allowing ordinary diff and merge operations on those files.

[6] "Sparkleshare - sharing work made easy," (http://sparkleshare.org/), last accessed Nov. 2011.

The Sparkleshare homepage, a solution for syncing between devices using Git as a storage solution. Similar in purpose to Dropbox, it may be possible to use the Git integration with ownCloud as a backend for Sparkleshare; this provides a full-featured, actively-developed desktop sync client for free.

[7] "The git community book," (http://book.git-scm.com), 2011, last accessed Nov. 2011.

The Git Community Book, along with Pro Git, provides an excellent overview of the Git internals, including packfile formats, index files and the compression algorithms used. Packfiles are used to store compressed representations of Git objects, and are read on the server-side when utilising "smart" push/pull over HTTP. A detailed specification of the formats and compression methods is invaluable for writing unit tests that deal with low-level functionality. Incorrect reading and writing of the packfiles can result in repository corruption, along with a loss of data.

[8] P. Fimml, "fimml.at," (http://fimml.at/#glip), last accessed Nov. 2011.

Glip is a PHP5 library providing access to Git repository packfiles and loose objects. The rest of Git's functionality can be constructed from these foundations, and Glip has been utilised to build several prototypes. There are no dependencies on the Git binary, reading and writing are fully implemented and the code is very well implemented. This project served as initial proof that it is possible to interact with Git repositories from PHP.

[9] D. Kuhn, "Distributed version control systems," Master's thesis, Stuttgart Media University (HdM), 2010.

Section 3 of this Master's Thesis describes in detail how Git operates on a technical level. Section 4 provides a

detailed overview of the Git object model, along with a brief investigation into the transport protocols and the commit process. This document was very useful for the Version Control Comparison report.

[10] A. Tridgell and P. Mackerras, "The rsync algorithm," The Australian National University, Tech. Rep., 1996.

The rsync algorithm utilises an algorithm discovered by Andrew Tridgell for efficiently transmitting large amounts of data across high-latency communication links. The algorithm splits files into fixed size, non-overlapping chunks and computes two checksums for each: an ordinary hash, and a rolling checksum based on adler32. The rsync algorithm is extremely efficient for synchronising files effectively, requiring only one round-trip for messages between clients.