Visualizing commit propagation in the Git super-repository of Linux

by

Gaurav Singh Thakur
B.Tech., Himachal Pradesh University, 2011

A Master's Project Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Visualizing commit propagation in the Git super-repository of Linux

by

Gaurav Singh Thakur
B.Tech., Himachal Pradesh University, 2011

Supervisory Committee

Dr. Daniel M. German, Supervisor
(Department of Computer Science)

Dr. Sudhakar Ganti, Departmental Member
(Department of Computer Science)

Supervisory Committee

Dr. Daniel M. German, Supervisor
(Department of Computer Science)

Dr. Sudhakar Ganti, Departmental Member
(Department of Computer Science)

ABSTRACT

A distributed version control system (DVCS) such as Git allows software developers to collaboratively contribute to a project without being on the same network. Every developer can work on their individual contributions within their private repository, only accessible to them. They can then collaborate and merge their work with the work of other developers via public repositories. During the project development life-cycle, many public repositories may collaborate with one another. The collection of all the repositories in a project is referred to as the "super-repository" of the Project. However, only some of this work actually reaches the main or "blessed" repository of the project which is used to release the final product/application. Visualizing how commits reach the blessed repository by propagating through the super-repository of a project can give us some insights on software evolution and development practices.

By developing a web application, this project helps in visualizing how every merge reached the blessed repository of the Linux project in 2012. This visualization is in the form of a merge-tree and two major challenges had to be overcome to visualize this tree: avoiding intersections and overlapping between different repository branches; visualizing the commits that were superimposed over one another.

It was observed that the project followed a hierarchical development practice and some integrating repositories could be clearly seen from the visualizations.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

my supervisor Dr. Daniel M. German, for his continuous support, guidance, mentoring and for giving me this wonderful opportunity.

my parents Kashmir Singh Thakur and Kamlesh Thakur, for everything.

my girlfriend Poonam, for fueling my ambitions.

'Do or do not, there is no try.'
- Master Yoda

DEDICATION

I dedicate this to Professor Daniel, my family and friends.

Chapter 1

Introduction

A distributed version control system (DVCS) like Git allows software developers to collaboratively contribute to a project without being on the same network. DVCS takes a peer-to-peer approach to version control, as opposed to the client-server approach of centralized systems. Rather than a single, central repository on which clients synchronize, each peer's working copy of the code-base is a complete repository [1]. GitHub is a web-based Git repository hosting service. It offers all of the distributed version control and source code management (SCM) functionality of Git as well as adding its own features. It provides access control and several collaboration features such as bug tracking, feature requests, task management, and wikis for every project [2].

Developers using DVCS usually work within their private repositories which are only accessible to them. They can then collaborate/merge their work with the work of other developers via public repositories. Contributions to a source code repository that uses a DVCS are commonly made by means of a pull request. The contributor requests that the project maintainer "pull" the source code change, hence the name "pull request". The maintainer has to merge the pull request if he or she decides the contribution should become part of the source base [3]. During the project development life-cycle, many public repositories may collaborate with one another. The collection of all the public and private repositories in a project is referred to as the "super-repository" of the project. Pull requests make sure that only some of this work would actually reach the project's main or the blessed repository. Only the work in the blessed repository is finally released as part of the main product/application. Therefore, visualizing how commits reach the blessed repository by propagating through the super-repository of a project can give us some insights on software evolution and

development practices.

Linux is one of the most complex software ever developed and Git was created by Linus Torvalds to manage the development of Linux. For developing the Linux kernel, the repository of Linus Torvalds is the blessed repository of the project. Only the successfully developed features reach this blessed repository. Rejected code or the code under development does not reach the blessed repository. The repositories around the blessed repository serve the same purpose as branches in a centralized version control system. Only Linus Torvalds can make commits in the blessed repository of the Linux project [4].

This project aims at visualizing the commit propagation in the git super-repository of Linux since this project required collaboration of thousands of contributors. Visualizing the successful commits that propagated from the git super-repository of Linux into the blessed repository should be a perfect case study for research and to get insights on how large software projects like Linux are developed.

In this project, a web application has been developed for visualizing how every merge reached the blessed repository of the Linux project in 2012. This visualization is in the form of a merge-tree and two major challenges had to be overcome to visualize this tree: avoiding intersections and overlapping between different repository branches; visualizing the commits that were superimposed over one another. Through this visualization we can get a glimpse of the development practice that was followed for this project.

1.1 Structure of the Project Report

This section provides information on what each Chapter of this Report will discuss:

Chapter 1 gives an overview on how developers collaborate their contributions within a software development project like Linux using a DVCS like Git.

Chapter 2 talks about the problem being worked upon in this project and the related work.

Chapter 3 discusses the technical design of the visualization web application.

Chapter 4 explains the implementation problems in visualizing the data and how

they were resolved.

Chapter 5 evaluates the visualization and talks about its limitations and future work.

Chapter 6 concludes the purpose and implementation of the project.

Chapter 2

Problem and Related Work

Visualization is one of the most intuitive and effective ways to quickly get a glimpse about the behaviour of a data-set. This project aims at finding and implementing an "effective" way to visualize how commits propagated within the git super-repository of Linux and reached the "blessed" repository of the project. This visualization can help researchers in the field of software evolution to study how collaboration takes place in building large software projects. Since git was originally created for managing the Linux project, visualizing this project can be a perfect case study for research in the field of software evolution.

In 2012, Daniel M German, Bram Adams and Ahmed E. Hassan [4] conducted research and collected the data of all the activity that took place in the super-repository of the Linux kernel by a process they called "continuous mining". They discovered all the public repositories involved in the project and then queried each of these repositories to see what new commits they had. They also checked the commits that had disappeared from these repositories. They identified 1,660,205 different commits across 529 different public repositories. Among these, only 68,477 commits reached the blessed repository via 2103 merges.

Since the data-set containing the information of all the commits that propagated through various public repositories into the blessed repository of the Linux kernel in 2012 has already been collected by Daniel M German et al., the purpose of this project is to find an effective way for visualizing this data-set, which can be found at: http://turingmachine.org/2015/linuxGit

Evan Wilde and Daniel M. German [5] proposed a way to visualize this data by generating a merge-tree with its root being a merge in the blessed repository. This project aims at improving this solution and generating a merge-tree for all the 2103

Field	Name	Description
Commit ID	cid	The ID that uniquely identifies a commit or a merge
Blessed Merge ID	mcidlinus	The commit ID of the merge where the current cid will be merged in the blessed repository of Linus Torvalds
Commit Date	comdate	The date and time of the commit
Next Commit	mnext	The commit ID of the next commit for a given cid
Next Merge	mnextmerge	The commit ID of the next merge for a given cid
Repository	repo	The repository for a cid where the commit is made
Author	author	The author of the commit

Table 2.1: Table describing the data-set needed for visualization

merges in the blessed repository in 2012.

Table 2.1 gives the description of the data fields, with their corresponding names in the provided data-set, used to solve this visualization problem.

Chapter 3

User Design

3.1 Technologies Used

One of the effective ways for visualizing this data is by drawing a merge tree for every merge in the blessed repository. The final target was to develop an application that is easily accessible to potential viewers/researchers, and is easily scalable in future. Therefore, building a web application seemed a logical step moving forward.

One of the most popular visualization library today is JavaScript's D3 Library. JavaScript was chosen as the main development language to utilize the visualization capabilities of D3. MEAN stack(MongoDB, Express, AngularJS, NodeJS), a popular JavaScript based full-stack, was chosen so that the entire web application can be built using a single language, JavaScript (front-end and back-end). The development was done on Cloud9, a web based IDE which helps in setting up web development environments. This allowed to make the development machine-independent. GitHub was used for version control.

3.2 Data Pre-Processing

Daniel M. German et al. collected the data of all the activity that took place in the super-repository of the Linux kernel. This data-set contains some irrelevant information in context of this project as only the data that reached the blessed repository needs to be visualized. Therefore, the relevant data had to be recognized and extracted before proceeding with the implementation.

File Name	Sample Data
commits.json	{ “cid”: “733bbb7e1c3acb8fabc55595bf1df8973dde7736”, “comdate”: “2012-01-01T19:36:08-08:00”, “log”: “Merge git://git.kernel.org/pub/scm/linux/kernel/git/davem/net” }
commitInfo.json	{ “mcidlinus”: “733bbb7e1c3acb8fabc55595bf1df8973dde7736”, “mnextmerge”: “733bbb7e1c3acb8fabc55595bf1df8973dde7736”, “mnext”: “fe3c8cc9226c7487c053edad9229dc85e93534d7”, “cid”: “cd3109d23c32452c85d73cc1a01282846a23582c”, “comdate”: “2011-12-30T14:15:41-08:00”, “repo”: “net/net”,“mwhen”:“2012-01-01T19:36:08-08:00”, “author”: “yevgeny petrilin ¡yevgenyp@mellanox.co.il¿” } , { “mcidlinus”: “733bbb7e1c3acb8fabc55595bf1df8973dde7736”, “mnextmerge”: “733bbb7e1c3acb8fabc55595bf1df8973dde7736”, “mnext”: “52793dbe3d60bd73bbebe28b2bfc9f6b4b920d4c”, “cid”: “fe3c8cc9226c7487c053edad9229dc85e93534d7”, “comdate”: “2011-12-30T20:32:45-08:00”, “repo”: “net/net”, “mwhen”:“2012-01-01T19:36:08-08:00”, “author”: “florian zumbiehl ¡florz@florz.de¿” }, { “mcidlinus”: “733bbb7e1c3acb8fabc55595bf1df8973dde7736”, “mnextmerge”: “733bbb7e1c3acb8fabc55595bf1df8973dde7736”, “mnext”: “c121638277a71c1e1fb44c3e654ea353357bbc2c”, “cid”: “52793dbe3d60bd73bbebe28b2bfc9f6b4b920d4c”, “comdate”: “2011-12-31T07:06:29-08:00”, “repo”: “1984-ayuso/net”, “mwhen”:“2012-01-01T19:36:08-08:00”, “author”: “julian anastasov ¡ja@ssi.bg¿” } , { “mcidlinus”: “733bbb7e1c3acb8fabc55595bf1df8973dde7736”, “mnextmerge”: “733bbb7e1c3acb8fabc55595bf1df8973dde7736”, “mnext”: “733bbb7e1c3acb8fabc55595bf1df8973dde7736”, “cid”: “c121638277a71c1e1fb44c3e654ea353357bbc2c”, “comdate”: “2011-12-31T07:59:04-08:00”, “repo”: “1984-ayuso/net”, “mwhen”:“2012-01-01T19:36:08-08:00”, “author”: “xi wang ¡xi.wang@gmail.com” }

Table 3.1: Sample data in an index for the two JSON files used in the project

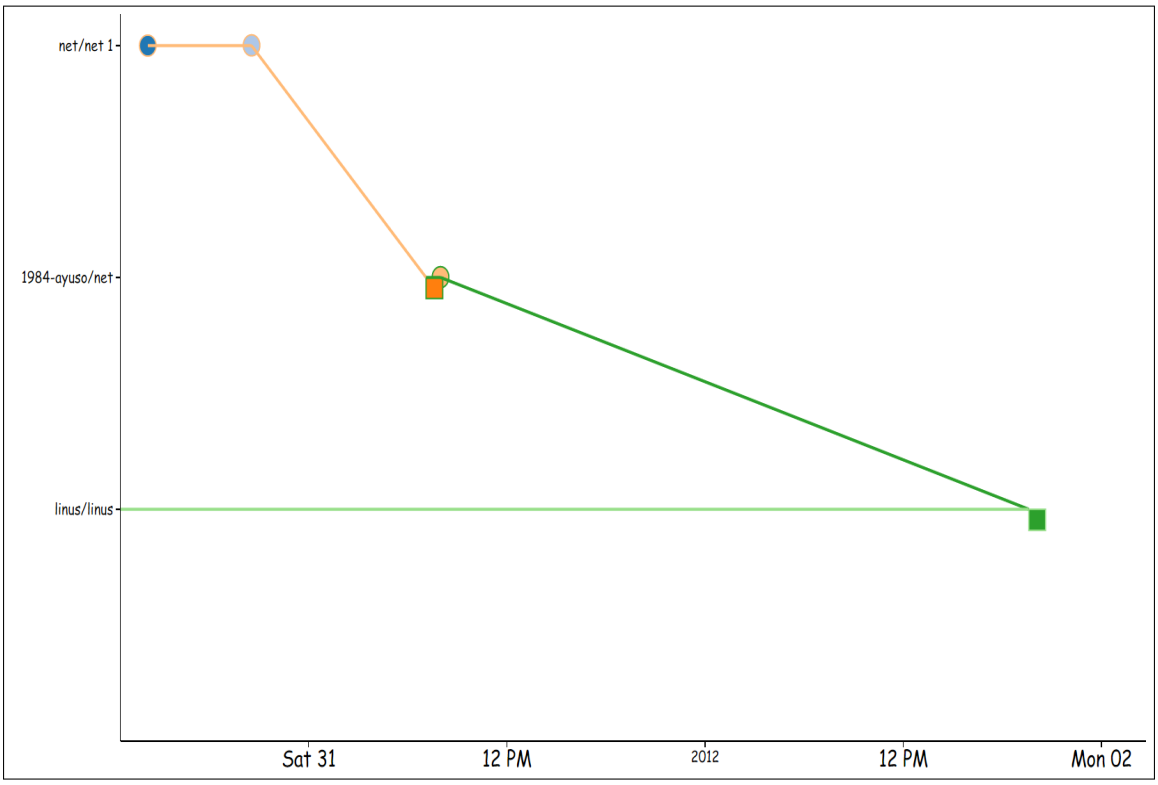

Figure 3.1: Merge-tree generated for the sample data in Table 3.1

3.2.1 Data Conversion

The original dataset was in the form of a 5GB postgres database which was too big to host on Cloud9 IDE free of cost. This dataset had commit information of all the commits made within the super-repository but we only required the information of the commits that reached the blessed repository. One way to resolve the problem of storage was to create a new database (subset of original database) containing the required data while the other option was extracting relevant data into JSON (JavaScript object Notation) files. For this project, the data was converted into a JSON format as it is easier to manipulate using JavaScript and takes much less storage space. It was possible to store the required data in a 30MB JSON file which reduced the original database size by a factor greater than 150.

JSON format helps in storing information within objects as a key-value pair. Two JSON files were used as input JSON data files. One file (named commits.json) contains the list of all the 2103 merges made in the repo of Linus Torvalds in 2012 with metadata about their commit id, commit date and log message. This file is used to display information in the homepage of the application as the end user can choose

to visualize the merge tree for any of these merges.

The detailed information of all the commits in the super-repository that were part of this merge is kept in another JSON file (named commitInfo.json). This file is used to draw the final merge-tree in the visualization page of the application. This file has an object commitInfo as the key with its value being an array of 2103 arrays. Every index of the outer array contains an array, with the information for a single merge in the blessed repository. The index value for the outer array in the commitInfo.json file is the same as the index value for a merge in the commits.json file. Every index of the inner array contains objects in the form of another key-value pair as can be seen in Table 3.1. This table shows the sample data in each index for the two JSON files. Figure 3.1 shows the generated merge-tree for this data.

3.3 Application

3.3.1 Homepage

The main page of the application consists of 2103 merges in the blessed repository of Linux in 2012. There is an option to dynamically search (while typing) for a specific commit by its commit ID or by its commit message. The users can also filter the commits by providing a date range. Once the user clicks on any commit its merge tree visualization opens in a new tab. Figure 3.2 shows the Homepage of the web application.

3.3.2 Visualization Page

The visualization page (Figure 3.3) opens in a new tab each time the user clicks on a commit from the homepage. The X-axis represents commit time while the Y-Axis shows the repository branch where the commit was made. Colour and shape are used for the following representation within the visualization:

1. A commit is represented by a circle
2. A merge is represented by a rectangle
3. Commits and merges are connected to each other by lines
4. The colour of the lines, outer border of the circles, and outer border of the rectangles represent the repository

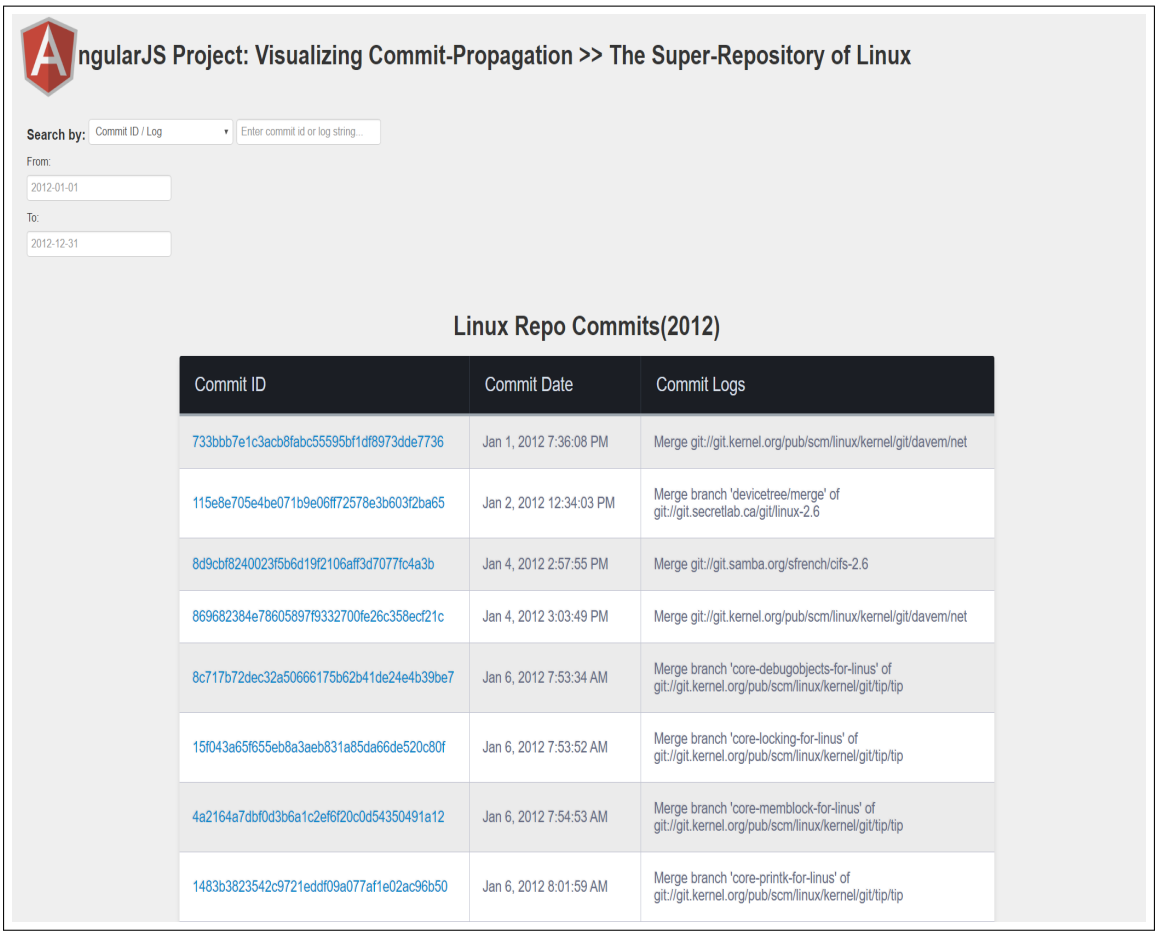

Figure 3.2: Homepage

5. The colour fill of the circles and rectangles represent the author

This visualization is zoomable along X-Axis (commit time) upto a precision of 1 second. The page also has legends on the right to show the colour coding of the respective author. The visualization is also responsive and all the information about a commit appears on the tooltip on hovering the mouse over it. The user can also click on any commit and the original commit page on github opens in a new tab. Figure 3.3 shows the visualization page for one of the merges.

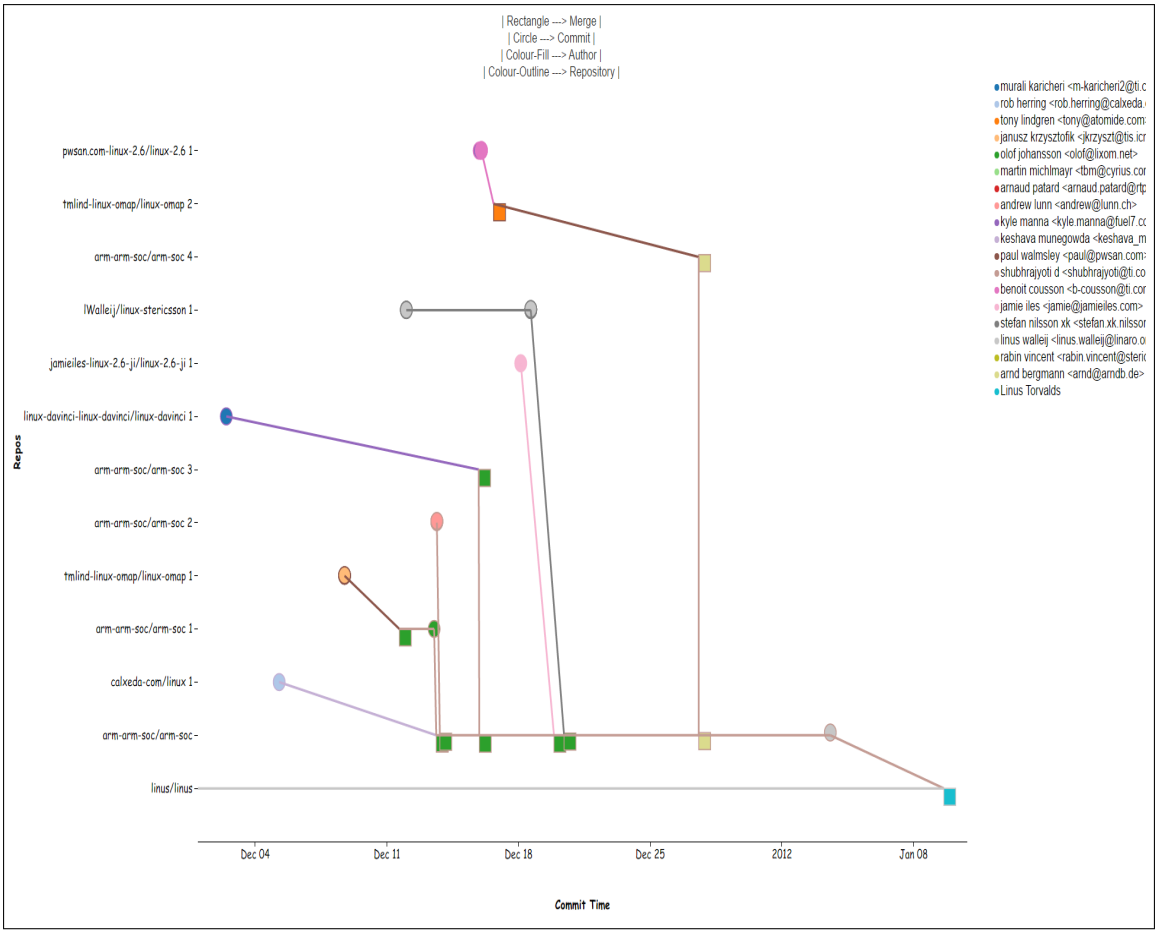

Figure 3.3: Visualization page

Chapter 4

Implementation

4.1 Application Architecture

The back-end of the application is built using NodeJS. The server code is present in the server.js file. The server can be started by simply writing the command "node server.js" in the terminal under its file location. It enables the application to run on the Cloud9 host at the URL: https://myproject-gauravsinghthakur.c9users.io. This URL loads the index.html file that contains most of the front-end code built using HTML, CSS, AngularJS and D3 Library.

AngularJS was used to build the application front-end. AngularJS uses a MVC (Model View Controller) Architecture to help build single page web applications (SPA). This architecture helps in interacting with and manipulating data without reloading the page. AngularJS allows us to define variables within a controller under scope variables. We can then use javascript to manipulate the HTML DOM (Document Object Model) within these variables. These scope variables can then be directly bound and used within the HTML tags of the web page. The Controller code, present in the index.html file, loads the data from the commits.json file in the Homepage. This controller contains a scope variable named $commitTree which is a method to draw the merge-tree for a merge in the blessed repository. When a commit is selected in the Homepage the Controller gets the data for that commit from the commitInfo.json file based on the index of the selected commit. This $commitTree scope variable is bound within another HTML file name commitInfo.html which loads in a new tab with the URL being the Homepage-URL/index of the merge.

4.2 Implementation Challenges

This project aims at effectively visualizing the merge-tree for a merge in the blessed repository showing how commits within the super-repository propagated and reached it. The visualization, however, was not as straight-forward as it first appeared.

The X-dimension of the visualization represents the commit time while the Y-dimension denotes the repository of the commit. Simply plotting all the data points along these two dimensions and connecting the next and previous commits together made the visualization chaotic (Figure 4.1), as there were intersections between the branches and some commits were super-imposed on one another.

There were the two major challenges faced in generating an effective merge-tree:

1. Every branch of the tree needed to be clearly distinguishable without any overlapping or intersections between one another.

2. We should be able to recognize commits that were indistinguishable due to there commit times being too close.

Figure 4.1 shows the result of the initial plotting of all the data points for one of the largest merge in the blessed repository. It clearly shows the overlaps. This problem was solved by developing a "merge-tree algorithm" and Figure 4.2 shows the visualization result for the same merge after applying the algorithm.

4.3 Merge-Tree Algorithm

The problem of overlapping only takes place across Y-Axis. We can logically reach this conclusion by the fact that we have no control over the relative positioning along X-Axis (as it represents time which we cannot change) but we can change the ordering of the repositories along Y-Axis.

However, it is impossible to build a merge-tree without any overlapping of the branches if the list of repositories along Y-Axis is unique along Y-axis. This is because multiple branches in this tree can have commits that occurred in the same repository. Therefore, those branches will move upwards or downwards (Figure 4.1) leading to intersections with other branches if we only have a unique line for each repository along Y-Axis.

This problem is dealt with by the merge-tree algorithm by creating new branches of the same repository for every merge in a repository. This data had to be created

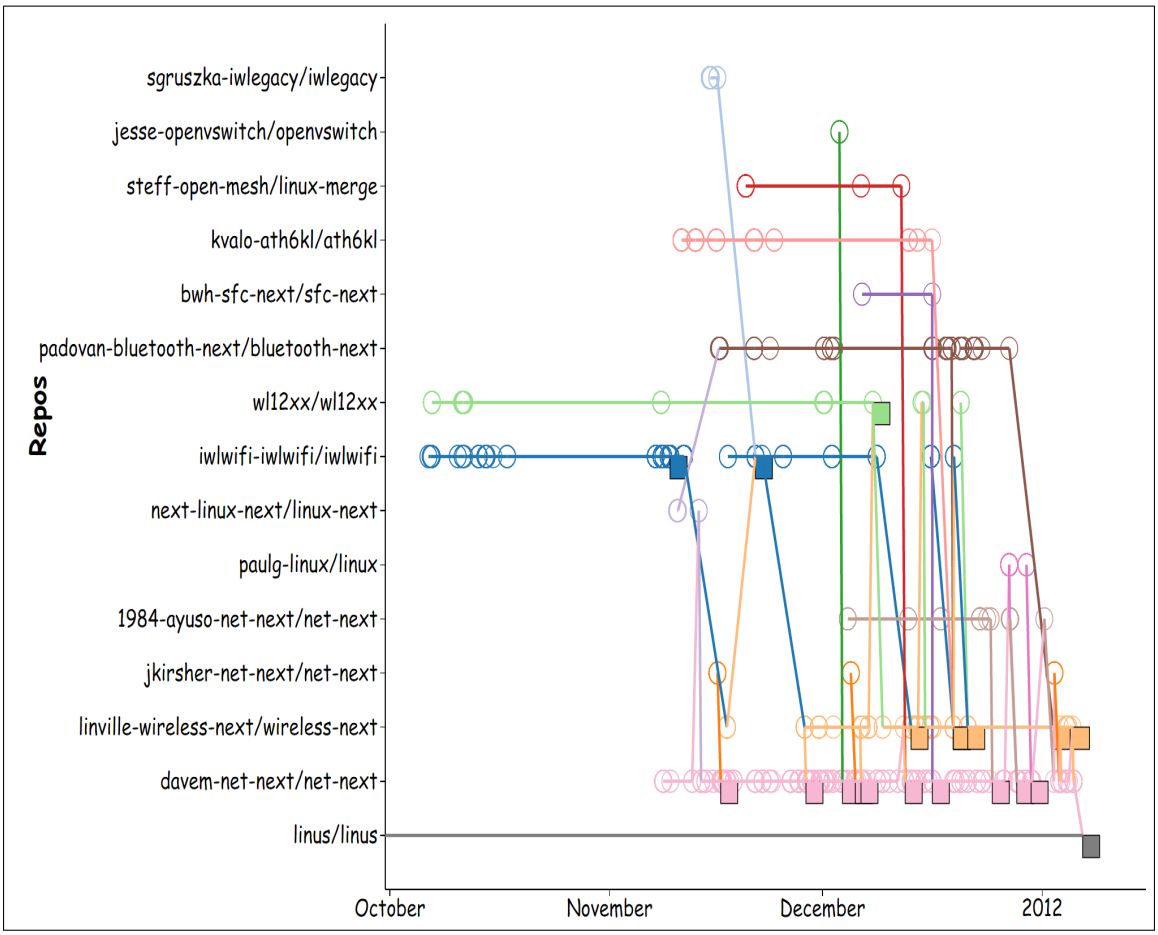

Figure 4.1: Visualization before the merge-tree algorithm

as there was no information about the repository branches in the original data set. Once we put the data in this context, this problem becomes a simple sorting problem along the Y-axis.

4.3.1 How does the Algorithm work

The merge-tree algorithm sorts the repository branches in a way that there is no overlapping or intersections in our visualization as can be seen in Figure 4.2.

Figure 4.3 shows the different system states and the possible transitions in the merge-tree algorithm. The merge-tree repository sorting algorithm works as follows:

1. A new array is initially created which will contain the sorted repositories. The first repository is set to linus/linus, i.e., the blessed repository of Linus Torvalds for the project, since this will have the root commit of the tree. The second

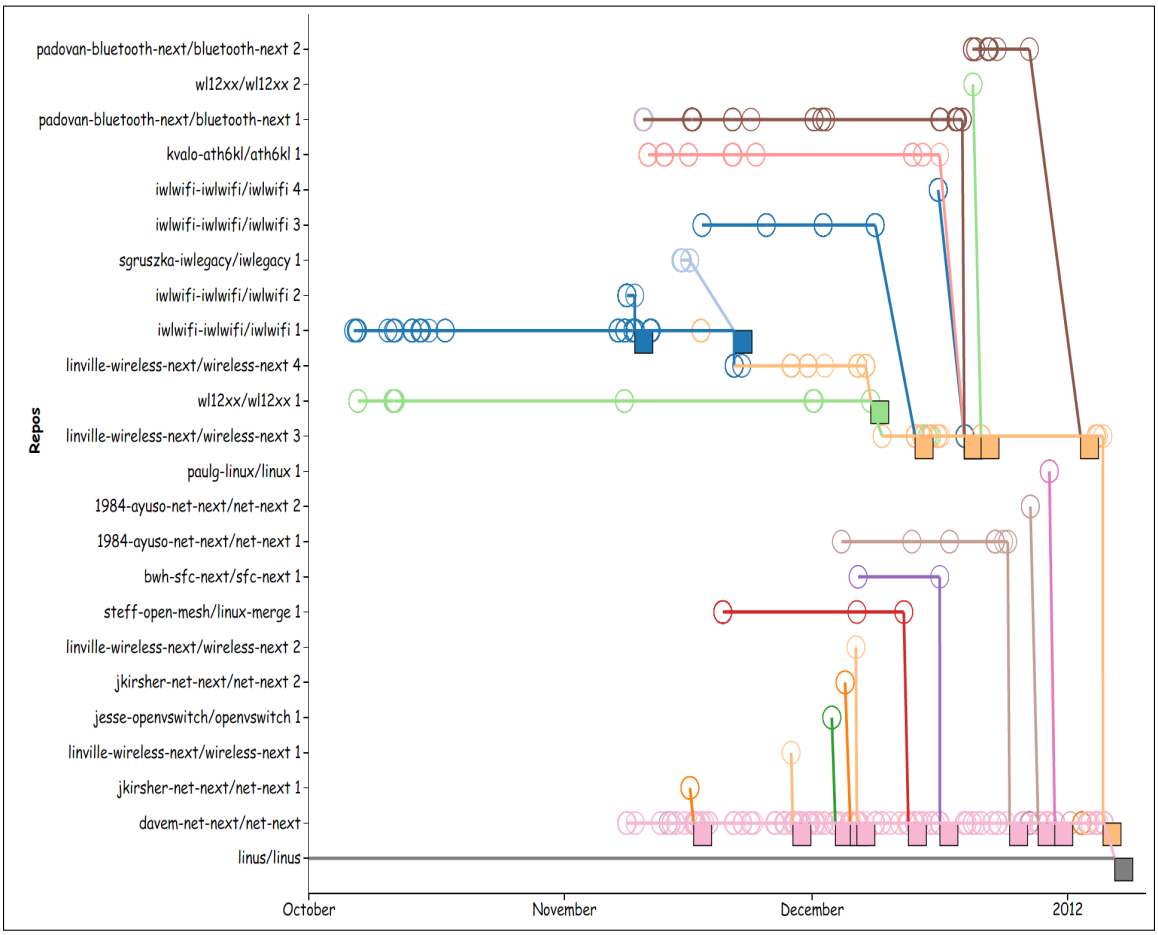

Figure 4.2: Visualization after the merge-tree algorithm

repository is set to be the repository of the previous commit of the merge in linus/linus as this will have the first child commit. At this point the system is in the "initial"state.

2. The algorithm will iterate through all the commits that reach the merge in the blessed repository one by one starting with the initial cid in the second repository. The system enters the "forward" state when the current cid is set to be the initial cid of the second repository. The system in the "forward" state keeps moving in the forward direction using the information in the mnext field for the next commit until it finds a merge. Once a merge is found the state of the system changes to "backward". The repository information for all these traversed commits is set to be the second repository. This is because all the fast forward commits would now appear in the second repository (with colour-outline of its repository) making the visualization clear.

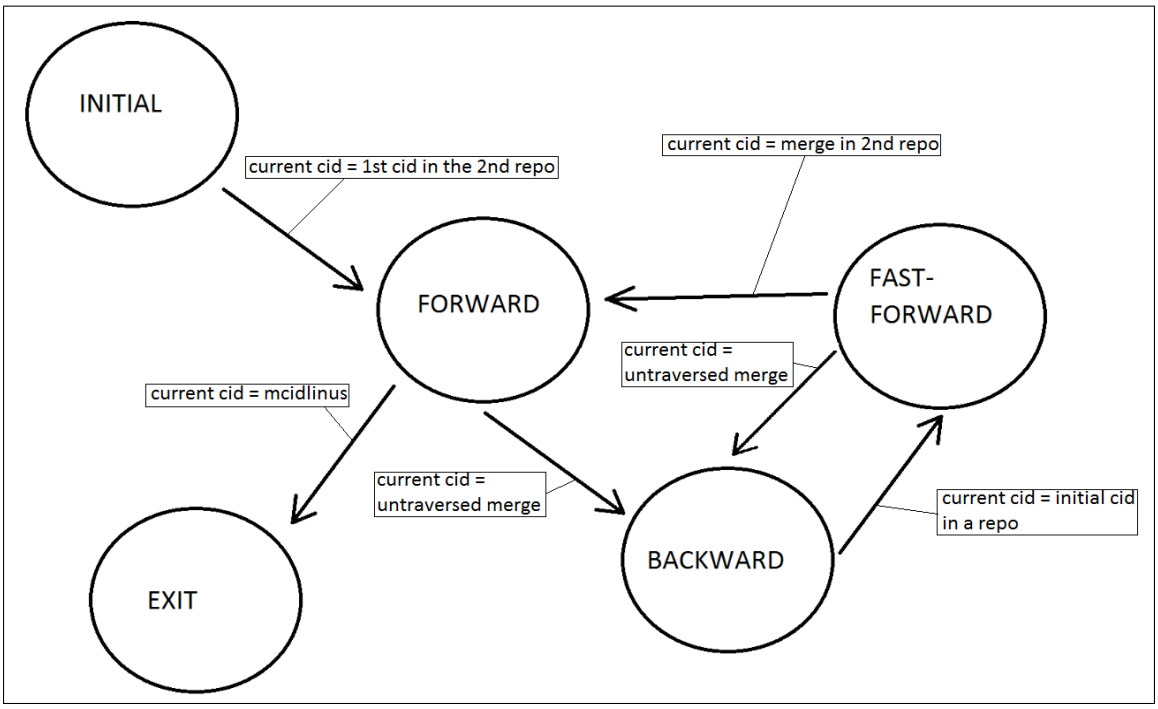

Figure 4.3: System states and the possible transitions in the merge-tree algorithm

3. In the "backward" state the pointer first creates a new repository branch with the repository of the previous commit of the merge. This repository branch is added next in the sorted-repository array. The system in the backward state keeps moving backwards to the previous commits until it reaches the initial commit in the repository. The repository information for all these traversed commits is set to be this new branch. Once it reaches the end/initial commit in a repository, the state then changes to "fast-forward". It should be noted that in the backward state does not care if the current cid is a merge and continues to move backwards until it finds a dead end. However, a merge usually (unless its an initial commit) has two or more previous commits and the previous commit chosen in the backward state is the one that is in the same repository.

4. In the "fast-forward" state the system starts moving forward again from the initial cid of the current repository branch. It stops at the next merge it finds whose branches have not yet been traversed. Once it finds such a merge, the repository of its previous commit is added to the sorted-repository array and the state now changes to "backward" again. It should be noted that the state changes back to "forward" once the current cid is the merge in the second

repository where the “backward” state first started.

5. The system reaches the “exit” state when the current cid in the “forward” state is the merge (mcidlinus) in the blessed repository of Linus Torvalds.

The pseudo code for the merge-tree algorithm is as follows:

```
// sorted_Repos is the function that returns a sorted array of
// repos as per the merge-tree algorithm.
// mcidlinus -> merge in the blessed repo
// curr_cid -> stores the current cid while iterating
// prev_cid -> stores the previous cid while iterating
// sortedRepos[] -> array containing the sorted repos
// id -> current id for storing a repo branch in sortedRepos
// flag -> current system state


function sorted_Repos(){
Step 1: Initialization
    prev_cid = previous_cid_to_mcidlinus;
    sortedRepos[0] = "linus/linus";
    sortedRepos[1] = repo_of_prev_cid;
    id = 2;
    curr_cid = initial_cid_in_sortedRepos[1];
    flag = "forward";

Step 2: Iteration
    While curr_cid is not mcidlinus {   // Exit State

        if flag=="forward {              // Forward State
            if curr_cid is a merge
                curr_cid = prev_cid_untraversed;
                flag == "backward";
                sortedRepos[id] = repo_of_curr_cid;
                id++;
                continue;
```

```
            else
                curr_cid = mnext;
                set repo of current cid as sortedRepos[id-1];
                continue;
        }

        if flag=="backward" {       // Backward State
            if curr_cid has no prev_cid
                flag = "fast-forward";
                continue;
            else
                set repo of current cid as sortedRepos[id-1];
                curr_cid = prev_cid_in_the_same_branch;
                continue;
        }

        if flag=="fast-forward" {   //Fast-Forward State
            if curr_cid is a merge in sortedRepos[1]
                flag = "forward";
                continue;
            else if curr_cid is an untraversed merge
                flag = "backward";
                curr_cid = prev_cid_in_the_untraversed_branch;
                continue;
            else
                curr_cid = mnext;
                continue;
        }
    }
}
```

4.4 Adding Jitter

Figure 4.2 shows that the merge-tree algorithm does a great job in removing the intersections and overlapping. However, it still does not guarantee that there will be no overlapping. We can intuitively conclude that an intersection is highly likely to take place when the commit time between a merge and its previous commit in a different repository are too far apart making the line between them much greater than 90 degrees. The merge-tree algorithm however removes most of the intersections.

The second challenge in the visualization was to make the commits hiding behind other commits visible (due to their commit times being too close). To resolve this issue the relative position of the commits that were super-imposed had to be offset.

This was achieved by creating a jitter between the commits that were committed within a time frame 1 second to each other by offsetting their relative positions (by 10% of the radius). Figure 4.4 shows the final visualization which included the colour coding, legends, merge-tree algorithm and the added jitter. The final visualization solves the issues of overlapping, intersections and gives a better idea about the volume of the commits. The visualization also provides a zoom feature which users can use to better view the high density regions and can get a better picture about the contributors.

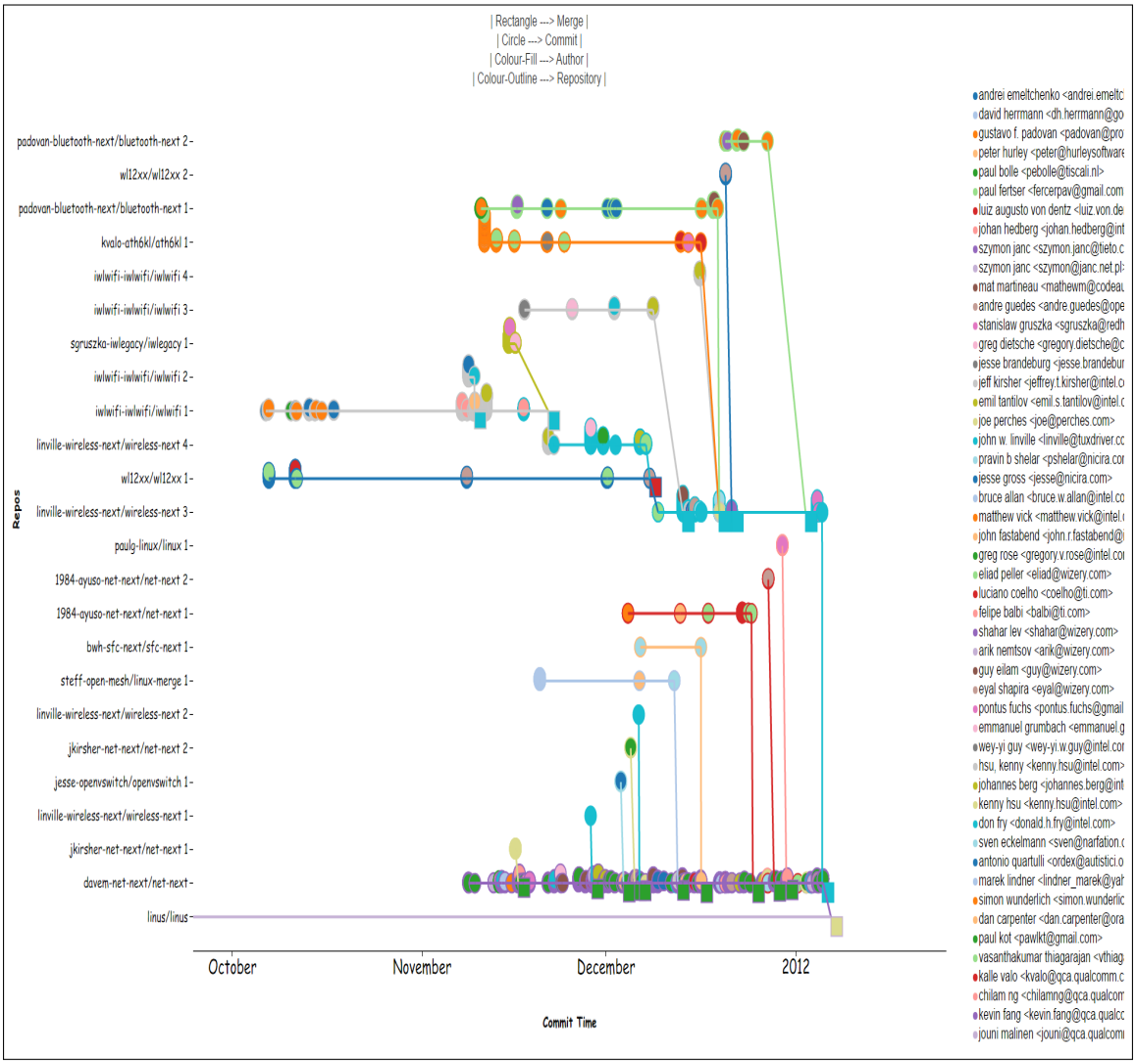

Figure 4.4: Final Visualization with jitter, merge-tree algorithm and colour coding

Chapter 5

Evaluation, Limitations and Future Work

5.1 Evaluation

This application helps in effectively visualizing commit propagation through various public repositories before finally merging into the blessed repository of Linus Torvalds. This can help researchers get insights on how collaboration takes place in large software development projects.

It can be observed from the visualization that the development for this project had a hierarchical structure. In other words, there were some integration repositories (especially the last one) where most of the development took place and these repositories were responsible for gathering all the code that would be later merged (through a pull request) into the blessed repository. There were other integration repositories which were responsible for gathering the code and merging it into the last integration repository, thus, creating a hierarchical development procedure.

Figure 5.1 shows some randomly selected merges that confirm this hierarchical nature of development and clearly shows how the final repository gathers all the code before it is merged into the blessed repository.

5.2 Limitations

Some of the limitations of this project are as follows:

1. The merge-tree algorithm does a great job in avoiding the intersections and

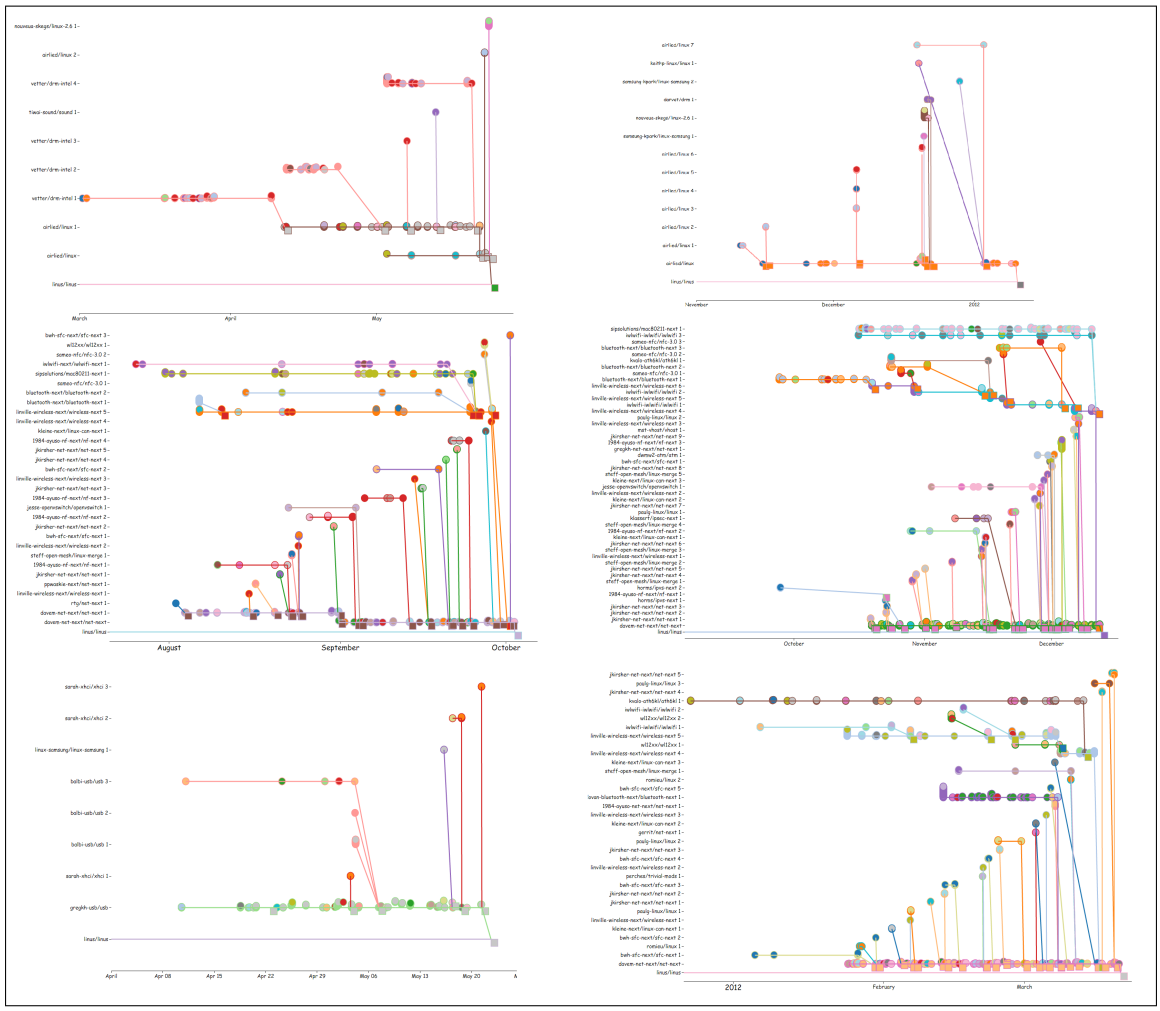

Figure 5.1: Visualizations showing a hierarchical development process and the presence of the final supervisory repository

overlapping. However, there would still be cases where we might see intersections. Intuitively, we can conclude that an intersection is highly likely to take place when the commit time between the merge and its previous commit in a different repository are too far apart. Some of the examples can be seen in Figure 5.1.

2. The data set used only has the information about the commits made in the blessed repository of Linus Torvalds in 2012.

3. The data used for visualization is currently stored in the form of a JSON file which is not as easily scalable as a database.

4. The colour coding used in the visualization can only take in 20 different colours, after which the colour codes repeat. Therefore, the visualization is not as effective if there are more than 20 developers who contributed towards a single merge in the blessed repository. In such a case, multiple developers will be denoted by a single colour.

   Nonetheless, on moving the mouse over a commit, the information appearing in the tool tip can help in distinguishing them. Also, there was no single merge where the repository count was greater than 20, so the colour coding works great to distinguish repositories.

5. The project is currently hosted on cloud9 at:

   https://myproject-gauravsinghthakur.c9users.io

   Since this is a free service, the server stops if it is inactive for more than an hour and needs to be restarted.

5.3 Future Work

Some of these limitations discussed can be easily overcome in future. Data for different years can be stored in different JSON files and the users can choose the data for a particular year.

The project can also be hosted on a paid cloud service (like AWS) which would allow it to be always available. On a paid service with high availability and high disk space, the data can be directly retrieved from a hosted database. This would make the application more scalable, and the data easier to maintain.

In future, this application can also be used to visualize the data that never made it into the blessed repository, as well as be used to visualize other projects for which the data-set is available.

Chapter 6

Conclusion

In this project, the goal was to build a web application to effectively visualize the commit-propagation in the git super-repository of Linux for the commits that reached the blessed repository of Linus Torvalds. This visualization is in the form of a merge-tree and can help researchers in the field of software evolution to study how collaboration takes place while building large software projects.

The two major challenges while implementing this solution were: how can we avoid intersections and overlapping between different repository branches; how can we visualize the commits that have been superimposed over one another. The former problem was solved by developing a "merge-tree algorithm" while the latter was solved by adding a jitter between the commits.

From the visualizations, it was observed that the development of project Linux followed a hierarchical structure with the presence of a few integrating repositories which gathered the code from other repositories.

It is noteworthy that this project is based on the research done by my supervisor Dr. Daniel German, who collected the data used in this project as well as proposed the visualization solution of generating a tree for every merge in the blessed repository. This project is an implementation with a few improvements to his research work. The published paper on his research is attached in Appendix A.

Appendix A

This Project is based on the following paper

A Dataset of the Activity of the `git` *Super-repository* of Linux in 2012

Daniel M. German
University of Victoria
Email: dmg@uvic.ca

Bram Adams
École Polytechnique de Montréal
Email: bram@cs.queensu.ca

Ahmed E. Hassan
Queen's University
Email: ahmed@cs.queensu.ca

***Abstract*—This dataset documents the activity in the public portion of the `git` *Super-repository* of the Linux kernel during 2012. In a distributed version control system, such as `git`, the *Super-repository* is the collection of all the repositories (repos) used for development. In such a *Super-repository*, some repos will be accessible only by their owners (they are private, and are located in places that are unreachable to other users) while others are available to other members of the team. The latter public repositories are used as avenues through which commits flow from one developer to another. During the last six weeks of 2011, we proceeded to automatically discover the public portion of the *Super-repository* of Linux. Then, in 2012, every 3 hrs, each of these public repositories was queried to see what new commits it had and what commits had disappeared from it using a process we call *continuous mining*. This resulted in the identification of 533,513 different commits across 451 different public repositories and how they propagated through the Linux *Super-repository*, including the repository of Linus Torvalds (i.e., the main repository of the Linux kernel). This information could help us understand how kernel contributors use `git`, how they collaborate and how commits are integrated into the Linux kernel and into the repositories of organizations that distribute the kernel.**

This dataset is at http://turingmachine.org/2015/linuxGit

I. Introduction

A team that uses a distributed version control system (D-VCS) must have at least one public repo (in this context, public means a repo that is readable to at least one more member of the team) and every developer must have at least one local repo (usually private, i.e., not readable by any other developer). We refer to the set of all repositories of a team as the *Super-repository* of the project. When the *Super-repository* includes only one public repo, a *Super-repository* acts like a centralized version control system. All the commits flow from the local repos to the public one and vice-versa, at the request of the owner of the local repository.

In practice, a team that uses a D-VCS will have one or more public repos and each developer will have one or more private repos. One of the public repos is designated as the *blessed* repo of the project, which one would find the most up-to-date branch of development. Other public repos are used to exchange commits between each other, side-passing, if necessary the *blessed* repo of a project. In general, the entire *Super-repository* will never be fully visible to anybody. Many repos will be private and live in locations that are only accessible to their owners. Other repos might be only accessible to a subset of the team (e.g. in an intranet).

Linux is a large, successful software development project. Just in 2012, we identified 4,575 developers improving it. Linus Torvalds, its leader developer also developed `git`. While `git` is becoming a popular D-VCS, it was originally built to satisfy the requirements that Linux had. Hence, one would expect Linux to be one of the projects (if not the project) that exploits the most the features of `git`. This implies that it is worthwhile to understand how the Linux kernel uses `git` and what impact `git` has in the development process of the kernel. To fully understand how `git` is used by Linux, one would need to know how repos interact with each other and how commits are moved by developers from one repo to another. Such a study faces a plethora of challenges:

- The private repos in the *Super-repository* are unavailable to others.
- `git` has no centralized logging mechanism that documents who is creating a new repo and its location.
- At any given time, there is no information that documents what repos form the public portion of the *Super-repository* of Linux. While several servers host Linux kernel repos (such as kernel.org and GitHub), many others are spread around the world on servers of different organizations.
- The *Super-repository* continuously evolves. Over time, repos are created, destroyed, and moved.
- Neither repos nor commits record information about where commits were created or which repos they have passed through. Given two different branches in two different repos, once these two repos merge the changes from the other, the two branches are indistinguishable from each other. Merge commits (if they are created) might hint to the origin of the commits they have merged, but this information is not always recorded or it might be overridden.

To overcome these challenges, we have developed a method to mine the *Super-repository* of Linux, which we call *continuousMining*. Using *continuousMining*, we mined during 2012 the public portion of this *Super-repository*. This paper documents the resulting data, including:

- The URI of 451 public repos that contributed commits to the *Super-repository*. Every 3 hours, we scanned each repo looking for changes (new commits or deleted commits). We performed 31,336 repo-scans where the repo

had changed (an average of 70 scans per repo in 2012) and retrieved the corresponding changes.
- We identified 533,513 commits that were created in 2012 (485,027 non-merges and 48,486 merges). To put them into context, if one were to mine, at the end of 2012, the *blessed* repo of Linux (*blessed* for short), one would have found only 64,029 (8.3%) of them. The remaining 91.7% of commits did not reached *blessed*.
- The 533k commits were authored by 4,575 different individuals (using 5,541 different email addresses) and committed by 1,058 different individuals (using 1,172 different email addresses).
- The 533k commits contained 135,532 different patches.
- We identified 56 million commit propagations. A commit propagation is an event in which a commit is seen for the first time in a repo or disappears from it.

II. Description of the Data

In Linux, the repo of Linus Torvalds is the *blessed* repo of the kernel development. His repo serves as the destination where commits are expected to flow. However, his repo only tracks the successful code. Features that are being currently developed, or that do not make it to the kernel will never be seen in *blessed*. The main reason is that the repos around *blessed* serve the same purposes as branches in a centralized version control system: developers do their work in their local repos (their own personal branches) and only when it is ready, it starts to move towards *blessed*.

Blessed is only writable by Linus Torvalds. In Linux, commits move from the personal repos of their creators to *blessed* using a combination of email patches, pushes and pulls. A typical commit will be created in a personal repo, then emailed as a patch to a person responsible for integration (`git` keeps track of the metadata of the commit). Another alternative is for the creator to push her changes to her public repo (e.g., in github) from where the integrator can pull the changes into her private repo. This integrator will repeat the process: she will push the commit (along many others) to her public repo, and issue a pull-request to the next integrator (in the path to *blessed*). If the commit is deemed worth it, it will eventually reach *blessed* (Linus will pull the changes from the integrator public repo into this own private repo, and then push these changes to *blessed*).

Unfortunately, *blessed* contains no information about these interactions between repos. The only trace of these interactions are merge-commits (i.e., commits that combines the work of one or more branches into another branch), but not every merge results in a merge commit (e.g., fast-forward commits) and sometimes the log of the merge commits does not document that the commit performed a merge. Even if there is a merge commit, this merge commit only documents an actual merge operation, and does not record every single repo in the path from its creation to *blessed* (this path usually consists of fast-forward commits).

To fully understand how Linux uses `git`, we need to mine the entire ecosystem of repos. However, we will never have access to the private repos of developers. Fortunately, because the kernel uses pull-requests to move commits between integrators, almost every developer has have a publicly accessible repo where she can share commits with other developers. If we were to mine all these public repos of the team, we can get a better picture of how integration is done in the kernel. Furthermore, as mentioned before, once a commit moves from one repo to another, we cannot tell in which repo it appeared first, hence the propagation of commits between repos needs to be dealt with afterwards.

To address these issues, we have developed a method of mining D-VCS repos called *continuousMining*. *continuousMining* queries repos ar a certain frequency to identify commits that have appeared or disappeared since the last query. The details of the method, including its implementation, can be found in [2]. In that paper, we demonstrated that *continuousMining* is superior to querying the *blessed* repo at one time, since it is capable of observing code as it moves across repos, documenting when commits are rebased (a common operation in the kernel) and recording code that is not yet in the *blessed* repo (see Peril 4 in [1]).

To our knowledge, research on Linux has always used a single snapshot of the *blessed* repo of Linux. GhTorrent [3] continuously mines Github looking for new changes, but it does not record propagation of commits (when a commit appears in a repo, whether the commit was seen before in a different repo, and when). Hence, we are the first to document how commits propagate in a D-VCS.

III. Methodology: How the Data was Gathered

We started *continuousMining* on the Linux kernel in Nov 2011. By January 1st, 2012 we were mining 262 repos. This number grew to 530 by the end of the year. Every 3 hrs, for each of these repos: 1) we would synchronize our copy of the repo; 2) create a log of all commits (in all branches of the repo) and compare it to the previous iteration's log; 3) label any commit that was not in the previous iteration as "New", and every commit in the previous log no longer in the current one as "Deleted". [2] documents this process in detail. This dataset concentrates on commits created during 2012 (but also contains older commits).

For each repo, we needed to determine who its committers were. This was mostly a manual process. When we observed a new commit in a repo R, if the commit was not from a known committer to R, then there were two possibilities: R had a new committer (we searched the Web and mailing lists for evidence of this), or the commit originated in another repo S. If it originated in S, then either we were already mining S or not. To determine if S was a known repo, we looked at the every known repo T that the committer was allowed to write to. If the commit was found also in T during the same time window, then $S = T$ and we assumed that the commit originated in S and had propagated from S to R in the last three hours. Otherwise we tried to find S (which was not currently known) using any merge information in the repo, searching the mailing list and using the Web. We feel confident that for every commit

that reached blessed in 2012, we have properly documented its origin repo. This process is documented in [2].

IV. Data Schema

Name	Description
Commits	Metadata of commits.
Logs	Log message of commits
FilesMod	Metadata of files modified in commits.
CommitsBlessed	Commits found in *blessed* at the end of 2012.
Commits2012	Commits committed in 2012.
Merges	Commits that merged one or more branches.
Repos	Repositories mined.
Owner	Committers of a given repo.
Aliases	Unified names of developers and the email addresses they use (active in 2012).
PathToBlessed	The path in the DAG of *blessed* that describes how a commit reached *blessed*.
RepoProp	Propagation of commits: when a commit appears or disappears from a given repo.

TABLE I
TABLES IN THE SCHEMA AND THEIR DESCRIPTION.

The main entities of this dataset and their relationships can be described as follows: there are *developers* who have created *commits* in their public *repos*. A given *developer* (identified by his or her *uniname*) has one or more *email addresses* to identify oneself as the committer or author of a given *commit*. *Commits* propagate through *repos* from their *repo* of origin. Finally, *commits* flow from their *repo* of origin to other repos and ultimately into *blessed* via *merges*. A subset of *commits* have found their way to *blessed* (by the end of 2012–we call them *commitsBlessed*). Any *repo* has zero or more documented *committers* (we call them *owners*) who are the only developers who can commit to it. The schema of the database, depicted in Figure 1 documents these entities and relationships and Table I describes the purpose of each table.

A. Propagations

The table *repoprop* is composed of attributes *cid*, *repo*, *seen*, *op* (either 'N' or 'D') and *origin*. If *origin* is false a commit *cid* is either added (*op* 'N') or deleted (*op* 'D') from a given *repo* at date/time *seen*. If *origin* is true then this was the very first scan of a repo (*op* is 'N' for all the commits in this scan). Repos were scanned every 3 hrs. We choose a period of 3 hrs because it was long enough to complete a scan (most scans took between one and two hours, depending on network traffic) and most repos didn't change during this period. Because we could not stop activity in the repos during the scan, but a scan was not an atomic operation. In few instances new commits had propagated to more than one repo between scans, and we resolved this manually (see [2] for details).

B. Path to Blessed

The information collected in *repoprop* is similar to the propagation of a disease. We know where a commit originated, and, at every snapshot (at 3hrs intervals) we know other repos that also had the commit (or that deleted it). However, when a new repos receives it, we do not know for certain which repo it received it from. Given a set R consisting of two or more repos that had the commit c at snapshop t_i and a new repo $S \notin R$ that has c as t_{i+1}, S could have received c for any repo in R. For this reason, we combined information from *repoprop* and the directed acyclic graph (DAG) of the commits found in *blessed*. Because we know when commits arrive to a repo, we can convert the DAG into a tree, where each node (commit) has only one successor and only one merge into *blessed*. This tree is documented in table *PathToBlessed*: for any commit *cid*, its successor is *mnext*; its next successor merge is *mnextmerge* and it is eventually merged into blessed at commit *mcidlinus* on date *mwhen* (*mnextmerge* and *mcidlinus* can be the same). If the commit was committed directly into blessed (by Linus), *mcidlinus* is null. Figure 2 illustrates the use of this information. It shows the tree of commits that were merged into blessed at merge commit 5ede3ceb7b2c2843e153a1803edbdc8c56655950.

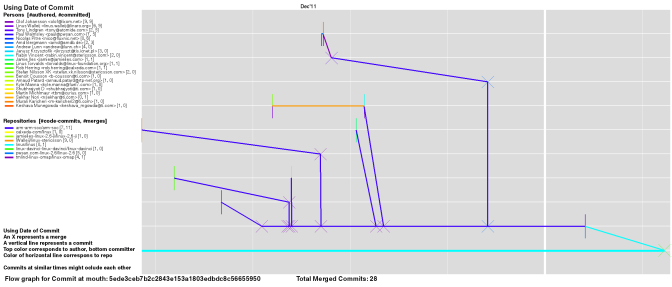

Fig. 2. Tree formed by the 28 commits merged into blessed at merge-commit 5ede3ceb7b2c2843e153a1803edbdc8c56655950. Each horizontal line represents a repo, Points marked with X are merges.

C. Patches in Commits

We found that as commits move throughout the *Super-repository*, they change commit-id [2]. This is because re-basing and changing the metadata of a commit are frequent operations used by Linux developers. For this reason we extracted the patch of every commit (`git log -patch`) and removed line-number context information; then we computed its hash, which we call the *codecontents* of the commit. While we observed 485k different non-merge commits in 2012, there were only 135k different patches. On average, the same patch appears in 2.3 different non-merge commits.

V. Threats to Validity

This dataset documents events that happened in 2012. While it contains events before 2012, these are incomplete. We used the period before 2012 to debug and calibrate our algorithms. For example, it contains repos that were inactive during 2012, and some propagations between repos before 2012.

Unfortunately some parts of this dataset are not reproducible. Once commits have propagated between two repos, it can be impossible to know where the commit originated. If a commit is rebased, it is very likely that the original commit is lost forever (unless the predecessor commit propagated to another repo; yet, it would be hard to know if this commit was the source of the rebased commit).

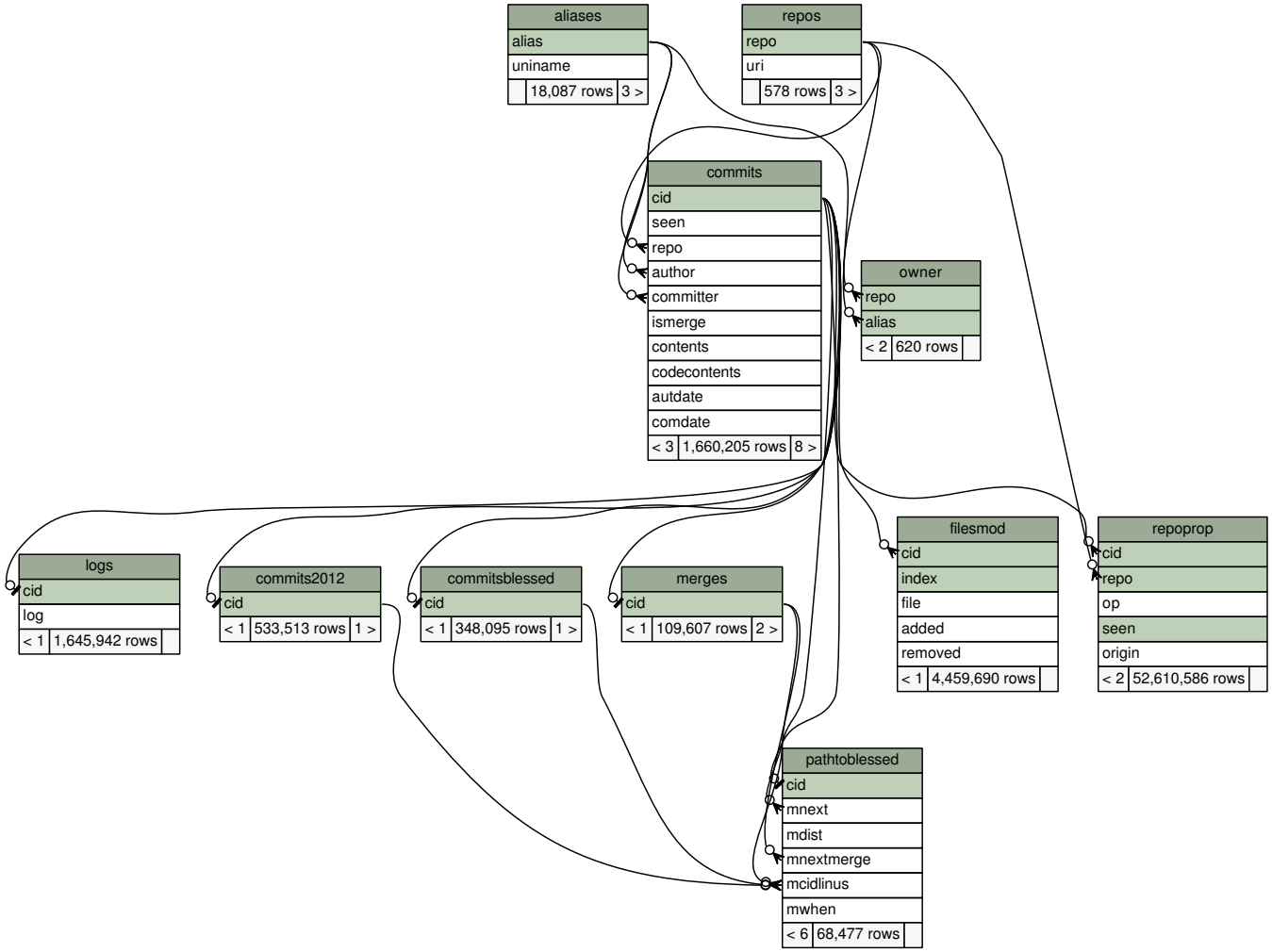

Fig. 1. Schema of the database. Shaded fields are the primary key of the table. To facilitate understanding, we have added foreign key constraints. The numbers under a table schema correspond to the number of tuples in it. The database contains 578 repositories, but only 451 contributed at least one new commit to the Linux *Super-repository* during 2012.

With regards to the data collection, we have done our best to manually verify it. We do not claim that we uncovered all the public repos that were active in 2012. We manually verified the source of all commits that reached *blessed* and were created in 2012. The unification of developers who committed or authored a commit in 2012 was done manually. We did not unify committers before (in that case, their *uniname* is null).

The window of 3hrs between the scan of a repo could have been too long. It is possible that, in between two scans, a public repo could have been updated—e.g., a commit is added— and in the next update such a commit has been deleted. In such a situation, the commit will not be recorded by us. We believe that although possible, such cases are unlikely, especially because the average time between updates of a repo was 5 days. When a commit propagated from its repo of origin to another repo between scans (i.e., the new commit is found in two new repos) we manually looked at the commit to determine its true origin.

Our dataset records all repos that we knew about and had access too. In some cases, we knew of the existence of repos but were not able to reach them (e.g., they were behind a firewall). Because we concentrated on repos that produced commits to *blessed* (as described above, we were diligent to find the source of commits that reached *blessed*) it is possible that we missed some repos that are never contributed back to the kernel (such as those of organizations that distribute linux versions) or those who had work in progress.

REFERENCES

[1] Christian Bird, Peter C Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. The promises and perils of mining git. In *MSR '09: Proc. of the 6th Int. Working Conf. on Mining Software Repositories*, pages 1–10, 2009.

[2] Daniel M. German, Bram Adams, and Ahmed E Hassan. Continuously mining the use of distributed version control systems: an empirical study of how Linux uses git. *Journal of Empirical Software Engineering*, To appear.

[3] Georgios Gousios. The GHTorent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR*, pages 233–236, 2013.

Bibliography

[1] David Wheeler. *Comments on Open Source Software / Free Software (OSS/FS) Software Configuration Management (SCM) Systems.* May 8, 2007.

[2] Alex Williams. *GitHub Pours Energies into Enterprise – Raises $100 Million From Power VC Andreessen Horowitz.* Tech Crunch. Andreessen Horowitz is investing an eye-popping $100 million into GitHub, 9 July 2012.

[3] Mark Johnson. *What is a pull request?* Oaawatch, 8 November 2013.

[4] Daniel M. German; Bram Adams; Ahmed E. Hassan. *A Dataset of the Activity of the git Super-repository of Linux in 2012.* Published in: Proceeding - MSR '15 Proceedings of the 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories Pages 470-473, 2015.

[5] Evan Wilde; Daniel M. German. *Merge-Tree: Visualizing the Integration of Commits into Linux.* VISSOFT 2016: 4th IEEE Working Conference on Software Visualization, 2016.