University of Passau

Department of Informatics and Mathematics

Bachelor Thesis

Analyzing Developer Networks Based on GitHub Issue Data

Author:

Raphael Nömmer

September 30, 2017

Advisors:

Prof. Dr.-Ing. Sven Apel
Chair of Software Engineering I

Claus Hunsen
Chair of Software Engineering I

Thomas Bock
Chair of Software Engineering I

Nömmer, Raphael:
Analyzing Developer Networks Based on GitHub Issue Data
Bachelor Thesis, University of Passau, 2017.

Abstract

Communication is a vital part of software development. To better understand how it influences the software development process, research is done using data from open-source projects and their communication channels. In this thesis, we look at a source of communication data that, to our knowledge, has not been used in network-based developer communication analysis: the GitHub issue tracker. To analyze issue-based communication networks, we extract the data from GitHub and integrate it into our library for network construction. We then construct issue-based author networks which we examine using several metrics. In a comparison to a mail-based network, we find that the issue-based network for OpenSSL and the mail-based ones share similar characteristics. When analyzing issue-based author networks for several open source software projects, we find that they show mostly consistent results. We conclude that the issue-based networks can be used for the analysis of communication among software developers, as an addition to, or a replacement for the mail-based networks that are in use now.

Contents

List of Figures

List of Tables

1 Introduction

A critical aspect of large-scale software projects is the collaboration among developers. According to Herbsleb et al., inadequate communication causes a lack of knowledge about the states of the separate parts of the project and can lead to issues not being addressed [HM01]. Furthermore, a lack of communication or poor communication is often stated as one of the main reasons for the failure of software projects [Cha05]. Since development, especially in open-source projects, has been becoming increasingly community-based and decentralized for several years now, new ways of communication have to be found, particularly in large-scale open-source projects where direct communication is not possible.

The first communication option that comes to mind are e-mails. They have been a widespread solution for a long time. However, there are other technical solutions that are becoming increasingly popular. The means of communication we are interested in are issue-tracking systems, which nowadays are more integrated into the software development process than e-mails. These tracking systems are implemented by tools such as BUGZILLA or OPENPROJECT. Online repository providers like GITHUB, BITBUCKET, or ATLASSIAN offer similar systems.

The goal of this thesis is to explore which new insights we can gain from analyzing issue data using network analyses. To achieve this, we are integrating issue data into an existing network building toolchain. The goal of this toolchain is to assist the analysis of the software development process in a network-based fashion by providing methods for network construction and examination. The toolchain offers ways to obtain data from git repositories and mailing lists.

In this thesis, we focus on data we retrieve from the GITHUB Issue Tracker. With over 57 million projects, GITHUB is one of the largest providers of online code hosting for private or open-source projects. In our analyses, we focus on five mid to large scale projects.

Before this thesis, the library that we use for network construction (which we call NETWORK LIBRARY from this point) only used one source of communication among developers: mailing lists. However, an increasing number of projects are using bug-tracking systems alongside mailing lists or using the issue tracker exclusively. Adding this data to the existing toolchain and analyzing the resulting networks will hopefully allow for further insights into developer communication and its impact on the project. Also, it is very likely that this data will become increasingly important as more and more developers are using GitHub and its features.

The integration of GitHub issue data into the toolchain consists of two main steps. First, we extract and process the data provided by GitHub. For the retrieval part, we use the functionality that the GitHubWrapper tool provides with some extensions. We have expanded this tool to increase its performance and support new features. We have also implemented post-processing on the data we obtain from the GitHubWrapper. Second, after retrieving the data and preparing it, we have to build networks for our analyses. For this purpose, we extend the Network Library. We add functionality to support the issue data. We also add network analysis so we can get new information on characteristics of the issue data - or, rather, the communication of developers and outside contributors in the issue tracker.

From the network analyses, we investigate five projects, one of which we discuss in detail including a comparison of the mail-based network and the issue-based one. We then compare the results of the five projects to see whether the results are consistent. We find that the issue-based author network behaves mostly similar to the the mail-based one and that most of the analysis results with the exception of the global clustering coefficient and the scale-freeness are comparable among the different projects or seem to be correlated to the network size. We conclude that issue-based networks are most likely suitable as an addition or an alternative to mail data depending on what is available though further research with an increased sample size should be conducted to confirm this.

The remainder of this thesis is structured as follows: In Chapter 2, we discuss terms and basics around the topic of this thesis that we use later. Specifically, we present some network-theory basics, then we take a look at GitHub and related papers. In Chapter 3 we explain the implementation of the new network type, issue-based author networks, and how we obtain the necessary data to construct them. The results of the network evaluation based on several metrics, are presented and discussed in Chapter 4. Here we have a look at the findings followed by an in-depth discussion thereof.

2 Background

We begin this chapter with a short introduction to networks where basics on network theory are introduced. We then take a look at GitHub, the development platform from which we retrieve the issue data that we are using to build networks. Finally, we take a look at related work.

2.1 Network Basics

Networks are a structure that is frequently used to study relationships among people or things. When they are used to analyze people and their mutual interactions, we call them social networks. In social networks, the actors represent the nodes and edges are represented by their relationship, communication or mutual activities. The data for social networks is mostly mined from social-media platforms, however, since we are focused on developer collaboration we use interaction data from software development to construct social networks. This form of social networks is also called developer networks [AJS11, LLH06, LLFGB04]. They will be examined in detail in Section 2.1.1.

To describe networks, we use notations from the field of graph theory. A graph $G = \{V, E\}$ consists of a set of nodes V and a set of edges E where each edge $e \in E$ connects two nodes in V. Graphs can be directed or undirected. In a directed graph every edge has a direction which means that $\langle x, y \rangle \neq \langle y, x \rangle$ where $\langle x, y \rangle$ is the edge from x to y. In an undirected graph those two edges are identical. A graph can also be simplified. When a graph is simplified all edges from one node to another are contracted into one edge if the network is undirected and into two edges if the network is directed, one for each direction. The number of edges that connect these two is the edge weight. If the graph is not simplified, there can be more than one connection between two nodes.

2.1.1 Developer Networks

When developers work together in open source projects, a lot of data about their communication and collaboration is generated. By connecting people involved in the same project that have communicated or have worked on the same parts of the project, we get a network. We refer to this kind of network as a developer network [AJS11, LLH06, LLFGB04]. Joblin et al. separates developer networks into two categories, developer coordination networks and developer communication networks [Job17]. Developer coordination networks are derived from data that is mined from a version-control system. The main part of this data is commit data. Using the commits, we can analyze how developers coordinate among each other and how it affects the progress of the project. Developer communication networks on the other hand are built using data from direct communication channels used by the developers. These direct communication channels could be mailing-lists or issues which we are using.

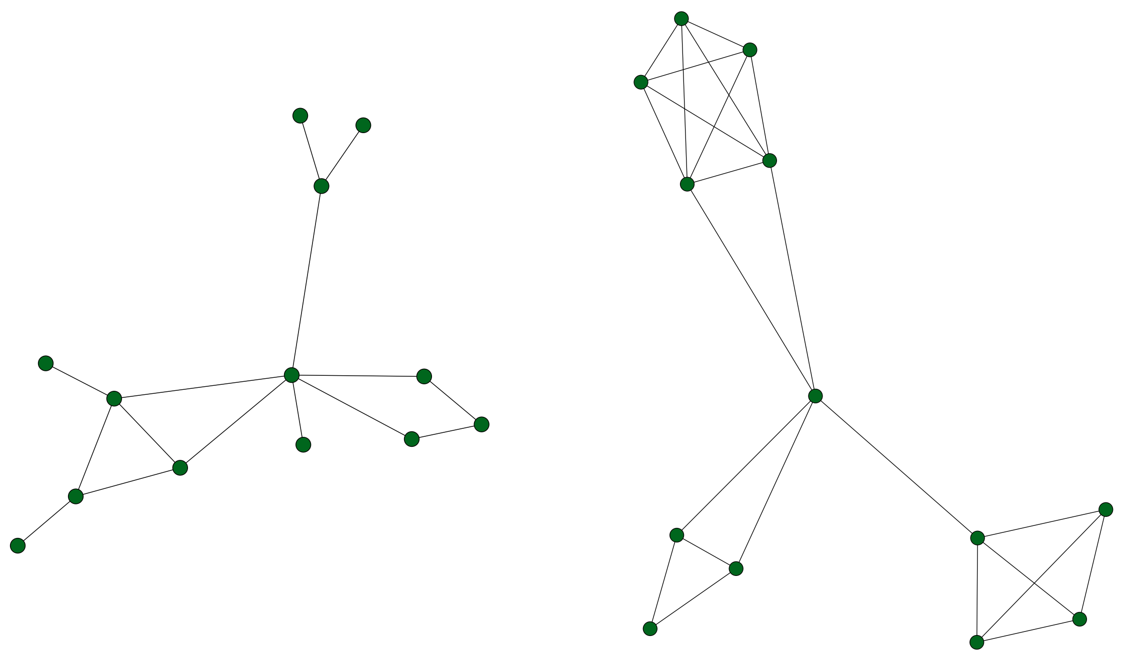
One thing we cannot deduce from the network is the quality of the communication. We cannot tell whether a conversation has importance to the project or whether it is completely off-topic. Still, it has been shown that developer communication networks of open-source projects are correlated with activities in the source code [XF14][BPD+08].

2.1.2 Network Metrics

In this subsection, we introduce and explain several network metrics which we will later use in the developer network analyses. First, we examine the more basic metrics. These are mostly concerned with the size and connectivity of the network.

The *number of nodes* is a simple way to get the size of the network and in our case the amount of people in the network. The *average node degrees* and the *maximum node degree* give a rough estimate of the networks characteristics. The degree of a node is the amount of edges that originate from or end in that node. The average path length is the mean length of the shortest path between every pair of nodes in the network. A path between two nodes v_i and v_j is comprised of the nodes and edges $v_i, e_1, v_2, e_2, ..., e_n, v_j$ with $e_k \in E, v_k \in V, k \in N$ and each edge e_k appearing no more than once. The shortest path between two nodes is the one that contains the least edges [BE05].

These were the basic network metrics, now we take a look at some more complex metrics that we use in the network analyses. The first of these complex metrics is the *clustering coefficient*. It is a number between zero and one that is low when the network shows no sign of groups forming and is close to one when it consists of groups the members of which are strongly connected among each other. There are different ways to calculate the clustering coefficient (also called transitivity). One way to calculate the so-called global clustering coefficient is $cc = \frac{3*\#triangles}{\#connectedtriples}$ where cc is the global clustering coefficient of the network n [WF95]. A triangle in this context is a construct of three nodes that are connected with each other while a connected triple is a connected subgraph that contains three nodes and two edges. Because a triangle can be seen as three different connected triples, this number ranges from zero to one. The other form of clustering coefficient is the local clustering coefficient. It can be calculated for a node by dividing the number of edges among the neighboring nodes of the node in question by the number of possible edges among the neighbors. From this, we can deduce the average local clustering coefficient which was measured by Watts and Strogatz and can be used as an alternative to the global clustering coefficient [WS98]. The drawback of this average local clustering coefficient is, that it ignores the node degrees. A node with only two neighbors has a clustering coefficient of one since all possible connections are present and it is weighted as much as any other node. In Figure 2.1 we show what networks with low and high clustering coefficient tend to look like.



Graph with low clustering coefficient | Graph with high clustering coefficient

Figure 2.1: With these two networks that have the same number of nodes, we show the difference between low and high average local clustering coefficient. The left graph has an average local clustering coefficient of 0.175. The network and the groups within the network are both loosely connected. The right graph on the other hand has an average local clustering coefficient of 0.803 which is caused by the tightly connected subclusters.

The next metric that we examine is *small-worldness*. A small-world network refers to a type of network in which the mean shortest path distance between nodes is sufficiently short as compared to the number of nodes in the network. This characteristic is implied by a high clustering coefficient and a small average shortest path length [BE05]. To quantify small-worldness, we use a method developed by Humphries and Gurney [HG]. Their approach compares the network of interest to a random network that has the same number of nodes and edges. For this comparison four values are calculated: the clustering coefficients C and C_r for the original and random network respectively and the average shortest path lengths L for the network to be analyzed and L_r for the random one. A value $S = \frac{\sigma}{\lambda}$ with $\sigma = \frac{C}{C_r}$ and $\lambda = \frac{L}{L_r}$ is then calculated from the clustering coefficients and average shortest path lengths. If $S > 1$, the network is categorized as a small-world-network. This means that either the clustering coefficient of a small-world-network is higher than that of a random one or the average shortest path length is smaller or both.

Scale-freeness is another one of the more complex network metrics. A network is called scale-free if its node degree distribution obeys a power law. When this is the case, there are few nodes with high degree and many nodes with low degrees. The Barabasi-Albert model explains the formation of scale-free networks with two main factors, growth and preferential attachment [BA99]. Growth means the network starts as a small network and additional nodes supervene over time. Preferential attachment means that there are few central nodes that have a lot of connections, also called hubs which are preferred candidates for new connections.

The second but last metric that we inspect is *network modularity*. While scale-freeness informs about individual nodes and their degree distribution, the connectivity of a node's neighborhood is ignored. The modularity metric defined by Newman and Girvan allows for quantification of said connectivity of the local neighborhood. We assume that the network that we analyze consists of k disjunct communities. Then we compute a $k * k$ matrix where an entry a_{ij} is the number of edges that links the subgraph i to subgraph j in the matrix. With this, network modularity is defined as $Nm = \sum a_{ii} - \sum a_{ij}^2$. Nm is close to one if the network modularity is high, that means the nodes within each community are well connected, it converges zero if the nodes within the communities are loosely connected. However, a value close to one is very rare, so a network has high modularity if Nm is in the range of 0.3 to 0.7 [NG].

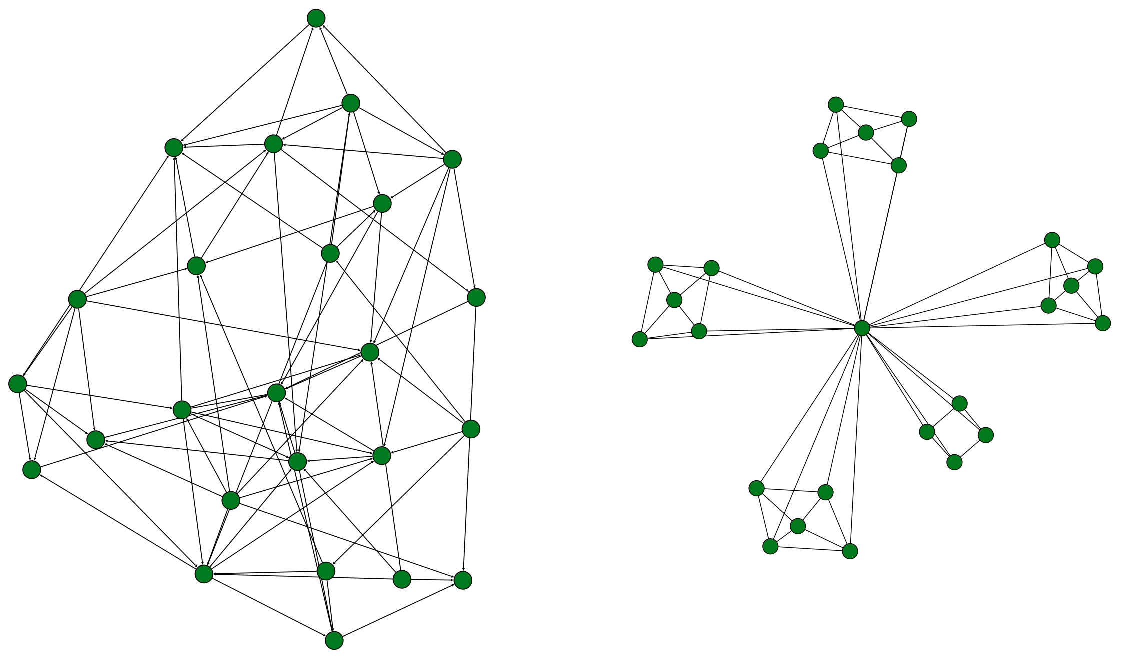
The final metric that we have a look at is *network hierarchy*. After looking at scale freeness and network modularity, which describe the distribution of edges amidst nodes and the grouping of nodes respectively, we now take a look at network hierarchy which combines these two concepts by looking at the relative arrangement of local groups. Hierarchical networks are characterized by a lot of nodes with a low node degree and high clustering coefficient and a few nodes with low clustering coefficient but high node degrees. These few nodes are on the top of the hierarchy, the others are ranked lower. In Figure 2.2 we show what a hierarchical network tends to look like in comparison to a random, non hierarchical one.

2.2 GitHub

GitHub is an online code hosting platform that offers remote repositories for Version Control Systems. It is git-centric but supports subversion as well. With more than 57 million repositories and over 20 million users reported in April 2017, GitHub is the largest source code hosting platform. It offers free hosting for public repositories as well as a paid service for private repositories. Aside from source code, GitHub offers several other features, for example Wikis, seamless code reviews, visualizations of several development aspects like commit frequency over time, and the feature which we are concerned with, the issue-tracker.

2.2.1 The GitHub Issue-Tracker

Issues are a means to keep track of tasks, enhancements and bugs for software projects. An issue in an open source repository can be opened by anyone and usually



Random graph Hierarchical graph

Figure 2.2: The network on the left is a random network that shows no hierarchy. The node degrees and local clustering coefficients are roughly uniformly distributed among the network. The graph on the right differentiates itself from the random network through small cohesive clusters within a larger less tightly connected cluster. It is hierarchical because there are few nodes with high degrees and low clustering coefficients and a lot of nodes with lower node degrees but higher local clustering coefficients.

contains a complaint or problem, a bug-report or a suggestion for possible changes or extensions. It contains information related to that itself like state, comments and commits to the version control system.

The issue-tracker allows a project's developers to manage a set of issues that everyone can see and aims to give an overview of the tasks at hand. The public access to the issue-tracker enables users of the software to report bugs and ask questions in a central place. Developers, besides answering those questions, can talk about the state of the project, discuss bug fixes etc. In comparison to mailing-list it is embedded into the development environment which means issues for example can be closed with a commit, people can be linked to an issue and will get notified and commits can be referenced to name a few of the features available.

For the purpose of this thesis, we include pull-requests with the issues. Pull-requests are a mechanism for developers to notify team members about changes done to a branch or fork and suggest a merge of those changes. In a fashion very similar to issues, the developers can then comment, assign people and reference people, issues, commits etc. The fact that GitHub treats issues and pull-requests very similar simplifies the inclusion of pull-requests a lot. If however we want to look at pull-requests or issues exclusively, they can be filtered using a flag that is part of the data that we get from GitHub.

2.2.2 Issue-Based Developer Networks

The type of networks, that we analyze in Chapter 4 are issue-based developer networks. As mentioned above developer networks use authors as nodes with interactions among them as edges. For the issue-based developer networks, an edge is constructed between two developers, when they have participated in the same issue.

While Joblin et al. use the mailing-list as an approximation of the complete communication among developers [Job17], issue-trackers usage is increasing as either a supplement to the mailing-list or as a replacement. This means that these networks allow us to capture communication that is ignored when solely looking at mailing-lists.

2.3 Codeface

Codeface is a tool for the analysis of software development projects. It can extract data from different sources such as version control systems, mailing-lists and bug-tracking systems. The resulting data is written to a MYSQL database, from which we then extract the data to CSV files. Codeface delivers a variety of data, but the parts that we are interested in and extract from the database, is the list of commits and the list of mails. For our analyses, we use it to obtain extract the mailing-list data, that we use for a comparison of mail-based and issue-based networks.

2.4 Related Work

Several papers up to this point have analyzed different aspects of the cooperation of software developers using mailing-lists. Some of them have also used a network-based approach like Joblin et al. [MJM17] who used mailing-list data to classify

developers into core and peripheral. The network-based analysis, in this case, was used to improve upon a count-based approach where only the numbers of mails, commits etc. were taken into account. Compared to the general analysis of issue-based author network that we perform, the paper by Joblin et al. had the goal of improving the classification of developers. Performing the same classification using issue-based networks and comparing the results might be of interest for future research.
Just like we do, Neumann et al. [DN10] look at issue-tracking data from a large software vendor. They use issue-based author networks as well, but they connect issues to software components and compare progress in the software components to the characteristics of the issue networks.
Rahman and Roy [RR14] investigate successful and unsuccessful pull-requests in 78 GitHub projects.

3 Implementation

In this chapter, we discuss the integration of issue data into the Network Library and the implementation of the network metrics that we use for evaluating these networks. The chapter is divided into four parts: The first part is to get the desired issue data from GitHub. To accomplish this, we use the GitHubWrapper tool. Next, we convert the issue data into CSV format which makes it easier to build networks later. At this stage, we also align the issue authors with the authors in the database from the Codeface analysis. This helps when building bipartite networks that are not build from issue data exclusively. Once the data is prepared, we can start with the construction of networks using the Network Library. The last part of the implementation are the network metrics that we use for our analyses.

In Figure 3.1 we give an overview of the whole process that is necessary to build issue-based networks. Here is a short explanation for each of the steps which are sequentially executed: (1) The GitHubWrapper tool requests the issue data from the GitHub API. (2) The GitHubAPI sends back the list of issues in JSON format and the GitHubWrapper tool adds data missing from the initial request. (3) The list of issues is saved on the disk in JSON format (4) The IssueProcessor reads the data from the disk (5) Users are verified using Codeface's IDService for the authors of the issues and comments. Users that were missing from the database are added. (6) The IDService returns the inspected users from the database to the IssueProcessor. (7) The IssueProcessor writes the resulting data to the disk in CSV format. (8) The CSV list is read from the disk by the Network Library. (9) The Network Library builds the network according to the parameters given by the user.

3.1 Extracting Issue Data from GitHub

The first step of the implementation is the retrieval of the issue data. GitHub offers two APIs for this purpose. One of them uses GraphQL, a query language developed by Facebook, the other one uses REST[1]. For the issue data extraction,

[1]GitHub REST API: https://api.github.com/

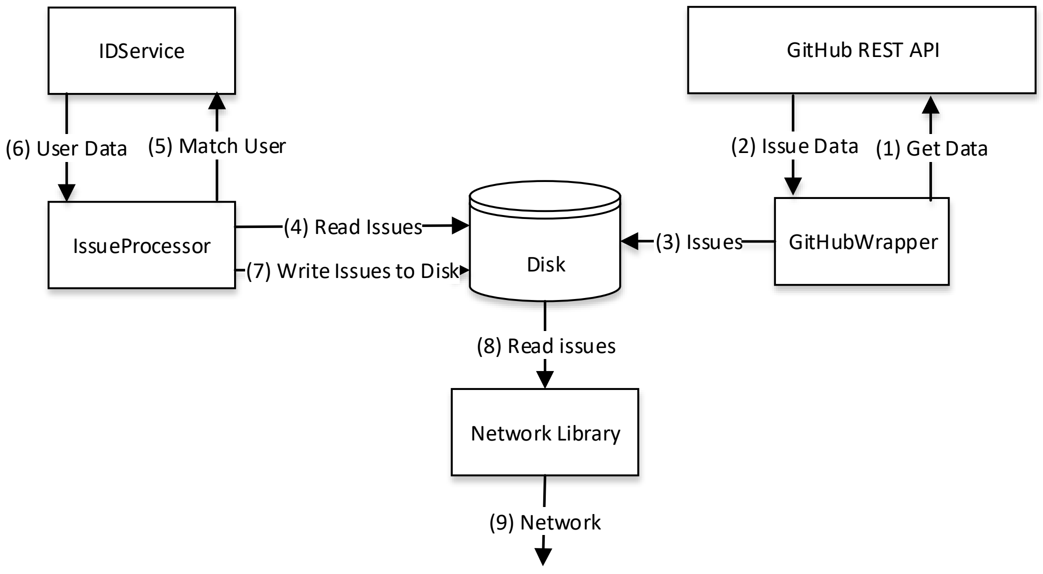


Figure 3.1: An overview of the issue extraction and network building process

the GitHubWrapper tool, which we utilize for obtaining the issue data, uses the REST API which can be accessed using HTTP-requests and provides data in JSON format. In this section, we first look at the GitHubWrapper in general and then at the two major challenges that occurred during its usage and how we dealt with them.

3.1.1 The GitHubWrapper

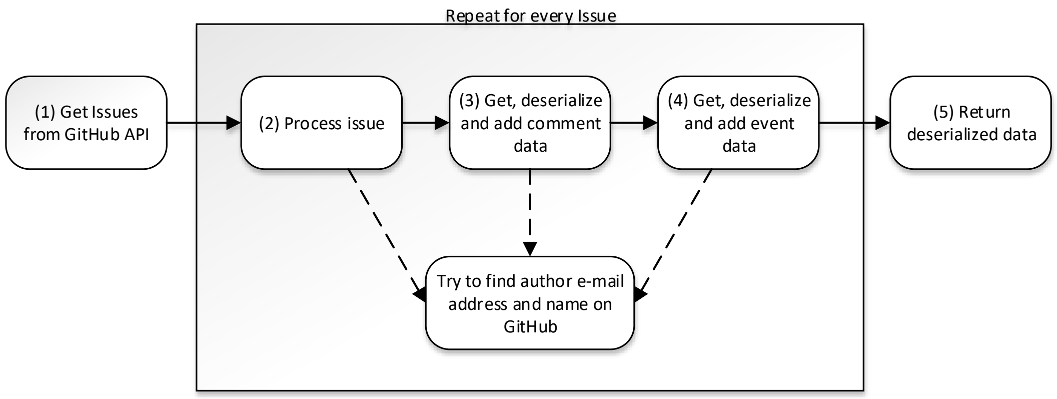

Repeat for every Issue
(1) Get Issues from GitHub API
(2) Process issue
(3) Get, deserialize and add comment data
(4) Get, deserialize and add event data
(5) Return deserialized data
Try to find author e-mail address and name on GitHub

Figure 3.2: Issue extraction with the GitHubWrapper tool

The GitHubWrapper[2] is a tool that offers a Java API for GitHub's REST interface, specifically for issues and pull-requests. It is an extension to the GitWrapper[3] library which provides a API around Git native calls. The GitHubWrapper tool uses Gson for JSON serialization and deserialization.

In order to use the GitHubWrapper, we have to add functionality for starting the process and saving the data to the disk, since the GitHubWrapper only provides a library that we can use. However, we will, for simplicities sake, still refer to the tool including the entry point we added as GitHubWrapper. The main method loads the authentication tokens which we will talk about more in Section 3.1.3, creates an instance of the GitHubWrapper's central class and makes the API call to retrieve the issues to said instance.

In Figure 3.2 we illustrate what happens when the call to retrieve the issues is made to the GitHubWrapper.

(1) The GitHubWrapper first makes the HTTP-request to get a list of all issues for the GitHub repository that is being analyzed. The next steps are repeated for each issue. (2) Then the tool deserializes the current issue from its JSON representation to a Java object to make editing it simpler. (3) Next, the GitHubWrapper makes a query to the GitHub API to get a list of comments for the issue that is being treated. These comments are then deserialized in the same way as the issues and added attached to the issue. (4) After that, the same is done with the event data. Events are all the things that can be done with an issue aside from commenting. This includes for example referencing someone in a comment, assigning a person to the issue or adding a label to the issue. This data is also attached deserialized and added to the issue. (5) The last step is to save the list of issues on the disk. For this purpose they are serialized back to their JSON representation and written to the disk.

3.1.2 Getting User Identification

The GitHub API only provides the username to identify a user with. Since, later in the process of preparing the issues for network construction, we need to match the authors found in the issue data with the developers found by Codeface in the analysis of the version control data, we try to get the e-mail addresses and the real name. For this, we attempt to get the e-mail address and name used by a person to make commits on GitHub. We can get a user's commit data by looking at his recent pushes. A significant disadvantage of this method is, that we can only get e-mails and names for users who have recently made code contributions to a project on GitHub. For users that are not developers in any projects or have not made any recent pushes, we have no possibility of getting the identification data.

3.1.3 Performance Improvements

Naturally, extracting issue data from GitHub is part of the critical path. The extraction has two considerable performance bottlenecks. The first of those performance limiting factors is the GitHub API access limit. To keep the degree of

[2]GitHubWrapper: https://github.com/se-passau/GitHubWrapper.git
[3]GitWrapper: https://github.com/se-passau/GitWrapper

capacity utilization under control, GitHub limits the number of requests that can be made to its API per user to 5000 requests per hour when using a GitHub access token. But 5000 calls still do not cover the number of requests that the GitHub-Wrapper makes to the GitHubAPI when analyzing larger scale projects. At least 3 calls are needed for per issue and big projects can have 10000 or more issues[4].

The second factor that slows the GitHubWrapper down, is the latency of the HTTP requests. The time it takes from making a call to the GitHubAPI until receiving the response ranges from around 400 to 800ms. This was tested using the bash to make the calls and measure the time they take.

To tackle the first problem, we use multiple authentication tokens. These tokens need to belong to different users, since the access limit is account bound. We also use pooling to manage the tokens. When all tokens are at their limit, we wait until there is a token available again. So the GitHubWrapper can work with one token but more tokens will significantly increase performance.

What multiple tokens also allow us to do, is making parallel calls to the GitHub API. We can run one thread per token since the limiting factors in terms of performance is mainly the HTTP latency and not CPU power. Another step to increase performance is to reduce the amount of HTTP calls we have to make by caching user data. Since the same user can appear a lot in different issues or within one issue this can reduce the number of requests that the GitHubWrapper needs to make.

[4]For example owncloud has over 12000 issues: https://github.com/owncloud/core

3.2 Processing Issue Data

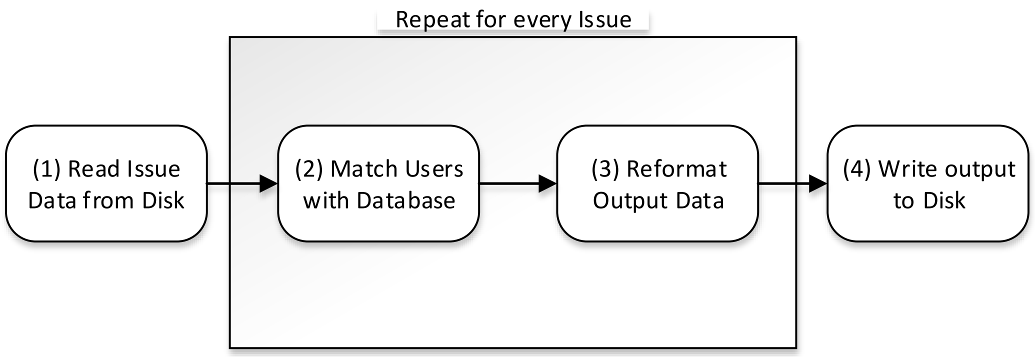


Figure 3.3: The activities performed by the IssueProcessor.

After the issue retrieval process is finished, the data needs to be transformed to make it suitable for the construction of networks in the next step. For this purpose, we have implemented a python script which we refer to as IssueProcessor, the procedure of which we depict in Figure 3.3:

(1) First the IssueProcessor reads the list of issues in JSON format. (2) In the second step, the IssueProcessor matches the authors found in the issue data with the developers that are in the Codeface database already. If an author does not yet exist, they are created at this point. This step makes sure that when building networks that use Codeface data and GitHub issue data, the authors can be matched correctly. (3) Now the IssueProcessor reformats the data to a CSV format, a table format where one line represents one event of an issue. This again makes the network building process simpler since it is in line with the other types of data, i.e., mails and commits. (4) Finally, the IssueProcessor writes the formatted data to the disk.

One problem that occurred at this point, was the difference in encoding between the IssueProcessor and the Codeface IDService which is the part of Codeface that we use for the retrieval of the author data in step (2). The Codeface IDService used latin1 encoding which caused problems with authors that have names containing characters that are not part of latin1. The existing behavior replaced unknown characters with question marks which could lead to multiple persons being treated as one leading to a falsification of the results. To fix this behavior, we changed the encoding of the whole toolchain, including the Network Library which also used to be limited to latin1, to UTF-8 and also employ UTF-8 as the output encoding for the IssueProcessor.

3.3 Network Construction

In this section, we discuss the construction of issue-based developer networks and how we implement it. Before looking at the integration of the issue data, we explain how the network construction process works with the example of mail-based developer networks. We use the same technique that is used for these networks for the new issue-based networks.

3.3.1 The Network Library

The Network Library is intended to build networks in order to examine the software development process. To do so, it uses data from Git repositories, mailing lists and, with the addition of issues, data from GitHub. With this data, several types of networks based on authors/developers, commits, mails and issues can be built.

The Network Library has two central classes, NetworkBuilder and ProjectData, which, as the names imply, are responsible for the construction of the networks and the preparation and handling of the data respectively. Additionally, two configuration classes manage the settings, where the data is read from, what type of network to build etc. There is a lot of complementing functionality like reading data, plotting networks, splitting networks or data by time etc. which is contained in several script files.

To get an overview of the network building process we will examine how a network with authors as nodes and thread contribution as edges is built using Figure 3.4: (1) First the user makes a call to get the author network from the outside to the NetworkBuilder instance (using either an R script or the R console). (2) The NetworkBuilder then requests the data from the ProjectData instance. (3) If the data has not been read yet, the ProjectData instance calls the read method for the mail data. This method changes depending on what type of network we want to build. (4), (5) The list of mails is read from the disk. (6) It is then translated into a data frame that contains one mail per row. Also, a standardized naming scheme is applied to the columns which allow for generic network construction. (7), (8) The data frame is then returned to the ProjectData instance where the mails are prepared for the network construction process. For this purpose, a list is created that contains the mails from the data frame split by the mail thread which they belong to. This means there is one entry for every thread with the thread id as the list entry's name and a data frame containing the mails in that thread as the values. (9), (10) This list is then returned to the NetworkBuilder. At this point the network construction itself begins. For this every thread is treated individually. For every thread, the authors in said thread are connected and the edge attributes are set according to the mails in the thread. (11) When the construction process is finished, the network is returned.

There are other networks with more than one node type. Bipartite networks can display the same relation as the author networks. A bipartite network with the mail relation for example connects authors to the mail threads they were a part of. The other network type with multiple types of nodes are multi-networks where the nodes are connected among the same type as well as with the other type.

Another feature of the library is splitting data. When splitting data, we divide it into several portions specified by time windows or by an amount of activity. This allows for the construction of several networks for one project. With these networks, we can look at if and how the development process of a project changed over time. The splitting can also be performed on networks after the construction process.

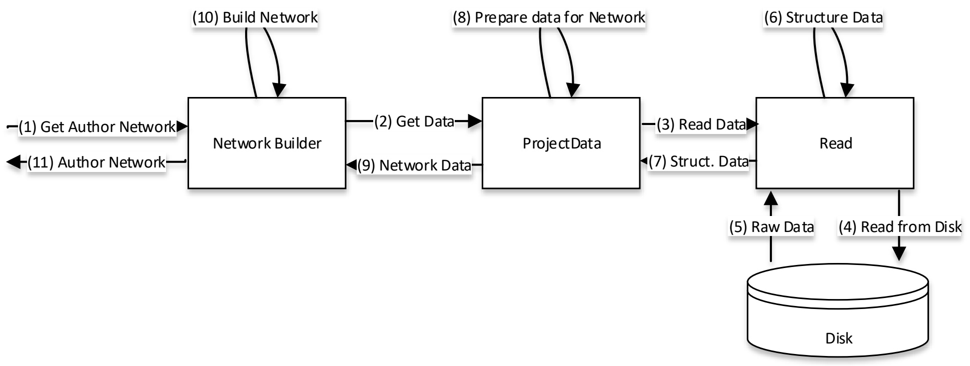


Figure 3.4: The Structure of the data extraction and preparation process

3.3.2 Integration of GitHub Issues into the Network Library

With the extraction and preparation of the issue data done as explained in Section 3.1 and Section 3.2, the integration of issues into the Network Library itself is fairly straightforward. The process of building issue-based author networks involves the steps shown in Figure 3.4 that we discussed above so we will not explain them again here.

To explain the network construction in detail, we take a look at some sample issue data, listed in Table 3.1 and how the corresponding author network is built. The sample data contains three issues. Every line represents one event in an issue each of which has an author and a date. In Figure 3.5 we display the network, built from this data. For every author in the issue data, a node is created. The nodes are then connected according to their activity in the issues. We have assigned one color to each issue in the graph. The issue with id one is represented by the red edges, issue number two is has green edges the edges for the last issue are blue. In Figure 3.5 we can see that all authors that work together on one issue are connected. The connections are created so for each event that a person has performed, edges are created to every other participant within the issue that the event is part of.

Some of the metrics that we apply to the networks, require the network to be simplified. This functionality is already provided by the Network Library. We show the simplified version of the graph in Figure 3.5 in Figure 3.6. The edges between two nodes are contracted to only one edge, in which the data that was previously distributed among multiple edges, is stored.

3.4 Network Metrics

For the analyses of the issue networks, we use the metrics that we discussed in Section 2.1.2. With these metrics, we aim to get an overview of the characteristics of

IssueID	AuthorName	AuthorMail	Date	Event
1	Adam	adam@gmail.com	2014-10-17 13:16:09	created
1	Clark	clark@gmail.com	2014-10-17 13:17:18	commented
1	Baker	baker@gmail.com	2014-11-19 20:24:19	closed
2	Evans	evans@gmail.com	2015-02-10 11:17:11	created
2	Davis	davis@gmail.com	2015-03-16 10:09:42	commented
3	Evans	evans@gmail.com	2015-10-17 13:16:09	created
3	Adam	adam@gmail.com	2015-11-24 18:07:03	commented
3	Baker	baker@gmail.com	2015-12-13 20:24:19	closed

Table 3.1: Sample issue data used for visualization of the network construction process reduced to the most important data

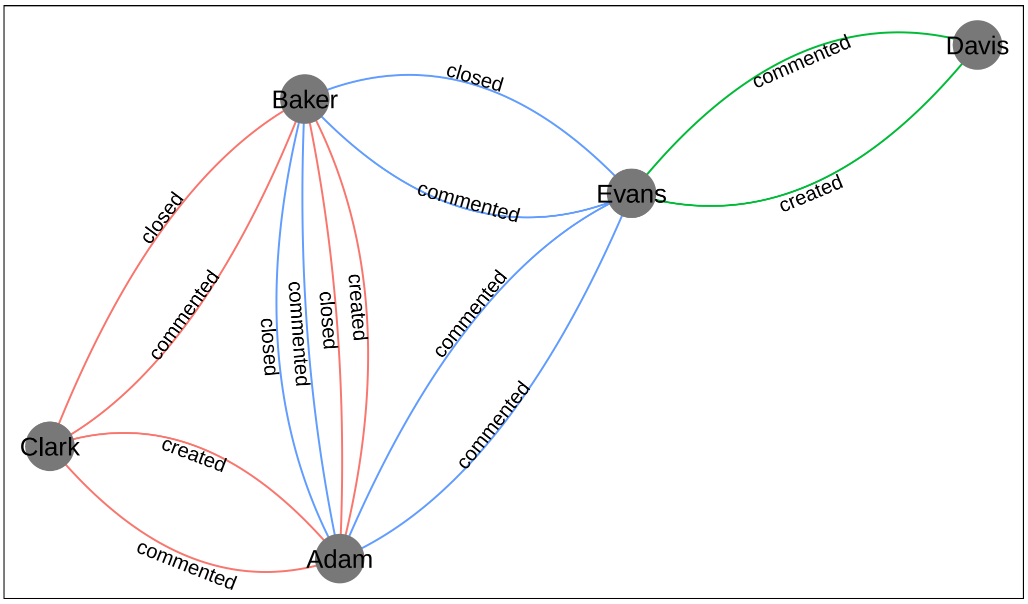


Figure 3.5: Issue-based author network constructed from the data in Table 3.1. The edge labels display the event that causes the edge. The color of the edges show to which issue they belong.

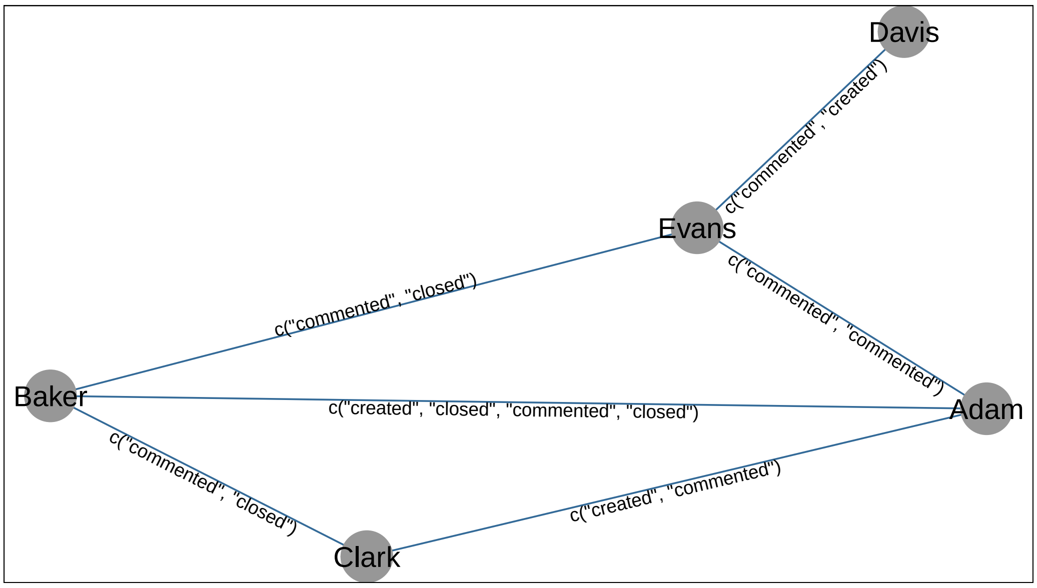

Davis
Evans
Baker
Adam
Clark
c("commented", "created")
c("commented", "closed")
c("commented", "commented")
c("created", "closed", "commented", "closed")
c("commented", "closed")
c("created", "commented")

Figure 3.6: Simplified version of the issue-based author network in Figure 3.5. The edges between two nodes are contracted to only one edge per pair of nodes.

issue-based author networks and whether they are suitable to be used in conjunction with mail-based networks for future analysis of developer communication in software projects. We aim to find out whether the issue-based networks deliver consistent results and whether the results networks behave similar to mail-based ones. Using the metrics, we can obtain information about the network and its underlying data. The implementation of these metrics was fairly simple due to the fact that IGRAPH provides implementations for most of the metrics.

4 Evaluation

In this chapter, we analyze issue-based author networks for five open-source projects using several network metrics. First we take a look at the projects themselves. Then we present the results that we got from the metrics and discuss their potential implications.

4.1 Subject Projects

We examine the following five projects: OpenSSL[1], Glasklart[2], Thimble[3], Brackets[4] and Owncloud[5]. In this section we will give some context on these projects to allow for a better interpretation of the experiment results. We now take a closer look at each of the five projects:

OpenSSL is a toolkit that implements the SSL and TLS transport layer security protocols. It also offers a general purpose cryptography library. With about 1,000 issues and 3,000 pull-requests that range from May 2013 to September 2017, the project is in the middle among the projects that we analyze in terms of size. Since we have mailing-list data available for OpenSSL which we have downloaded from the public mailing-list archive Gmane[6], we will compare the mailing-list-analysis results to the issue results for this project. The mail data ranges from September 2001 to February 2016, but we cut it the range of February 2008 to February 2016. The reason for this is explained in Section 4.2.1.

Glasklart is a theme for IOS. This project has around 9,200 issues and only 24 pull-requests which range from March 2012 to September 2017. Since they only have 64 people who have contributed to the source code of the project on GitHub at this time, this could either mean they use issues extensively or the project gets a lot

[1]https://github.com/openssl/openssl
[2]https://github.com/glasklart/hd
[3]https://github.com/mozilla/thimble.mozilla.org
[4]https://github.com/mozilla/brackets
[5]https://github.com/owncloud/core
[6]http://gmane.org

of issues from non-developers. Either of these possibilities might have an impact on the results of the metrics.

Thimble and Brackets are both code editors intended for use with HTML, JavaScript and CSS developed by Mozilla, though Brackets is forked by Mozilla from Adobe. The difference between them is that Thimble is a modified, web based version of Brackets that runs in the browser. Brackets has about 300 issues and 550 pull-requests which date from July 2014 to September 2017 at the time of the download and Thimble is at 1,300 issues and 3,000 pull-requests within the time frame from April 2013 to September 2017.

Owncloud is a cloud-hosting software. With around 16,000 issues and 12,000 pull-requests in the period from August 2008 to September 2017, it is the largest project that we analyze.

Project	Issues + pull-requests	Number of Developer	Number of Contributors (Issues)	Time frame
OpenSSL	4327	291	1354	2013-05 : 2017-09
Glasklart	9194	61	490	2012-03 : 2017-09
Thimble	2501	179	360	2013-04 : 2017-09
Brackets	877	315	95	2014-07 : 2017-09
Owncloud	29110	432	9554	2008-08 : 2017-09

Table 4.1: Overview of all the projects, their associated number of developers, issue contributors and the time frames within which our data is situated. The high developer count and low issue contributor count of Brackets is due to it being a fork.

4.2 Results and Discussion

In this section we present the results of the network analyses that we performed on the projects listed in Section 4.1. We first examine OpenSSL in detail and compare the issue results of the project with the mailing-list results. Then we take a look at the rest of the projects and see whether they show similar results for the issue-based networks or whether they show noticeable differences. We use undirected and simplified network for all the analyses that we conduct.

4.2.1 OpenSSL

Analysis of Issue-Based Networks

In Table 4.2 we show the simple metrics for the issue-based author network for OpenSSL. The maximum total degree is at 880 which in contrast to the average degree of roughly 7.9 tells us that some people participate in a lot of issues, while the majority is only involved in few issues. We can observe this structure in Figure 4.2 where we can see few central nodes with plenty of connections and a large number of nodes in between and around them with fewer connections. When looking at the total number of nodes and comparing it to the maximum total degree, we can see that at least one person has interacted with two thirds of the people that have contributed to the issues of the project. The density of the network is fairly low with 0.5% which means the whole network is connected rather loosely, which confirms that most people only contribute in few issues. Finally the number of nodes, which is the number of people who have contributed to at least one issue, is considerably higher than the number of people who have contributed to the source-code, which, at the time of writing this thesis, is at 291. This means that the issue tracker is used for questions and bug reports a lot.

Simple Metrics	Value
Number of Nodes	1354
Avg. Degree	7.926
Avg. Pathlength	2.284
Max. Total Degree	880
Density	0.00586
Number,of Contributors	291
Complex Metrics	
Clustering Coefficient (L)	0.919
Clustering Coefficient (G)	0.0484
Modularity	0.137
Smallworldness	15.074
Scale-Freeness	
alpha	2.397
x min	6
KS p	0.879

Table 4.2: Metrics results for the issue-based author network of OpenSSL. The (L) and (G) behind the clustering coefficient stand for the local and global clustering coefficient respectively.

Now we get to the complex metrics which we display in Table 4.2 and Figure 4.1. With a value of about 0.92, the average local clustering coefficient (marked with (L)) of the network is very high. This is most likely because the network has a lot of low degree nodes with connected neighbors, which can distort the result of this metric. However, the global clustering coefficient (marked with (G)) is fairly low. So the graph has a fairly low clustering behavior.
The modularity on the other hand is fairly low which means that the network does not consist of strongly connected subgroups with loose connections to the other communities. This fits with what we can see in Figure 4.2. Most nodes of the graph are nodes that are evenly scattered with a few central nodes. For the issue network this indicates that there are no groups within the project that handle issues among themselves but rather that people contribute to issues multiple topics in the project without strict topic limitations.
When calculating scale-freeness, we get three values. The most relevant one of them being *KSp*. It represents the p-value for the Kolmogorov-Smirnov test. If this value is below 0.05, a power-law does not fit to the degree distribution of the graph. Since the value for the OpenSSL results is well beyond 0.05, the network is scale-free. The other values tell us the exponent of the power-law distribution

(alpha) and the minimum node degree from which the power-law distribution fits the degree distribution of the network (x min). As described in Section 2.1.2, a scale-free network can be explained as a result of preferential attachment and network growth. Preferential attachment makes sense for this network because as we have seen, that there are developers that interact with a majority of people in the network. These nodes are favored for the connection of new nodes because these core nodes are the developers that respond to most of the issues. And the network has of course grown with the course of the development.

Next, we take a look at the small-worldness. Since the value for this metric is a lot higher than one, the network is considered a small-world network. It means that most nodes can be reached from any other node within a few hops. This categorization is caused by the issue-based author network being a social network which tend to be small-world networks. A lot of new issues that are created in a project, are responded by the few central authors which we can see in Figure 4.2. This keeps the average path length low, since the central developers are connected to a lot of other developers.

Finally we inspect the hierarchy graph in Figure 4.1. The nodes with a low position in the hierarchy are in the top left of the graph, the nodes at the top of the hierarchy are on the bottom right. There are a few authors that have a very high position in the hierarchy and many with a low position. This behavior can be seen in Figure 4.2 as well. There are few central nodes that are at the top of the hierarchy while most of the nodes are fairly evenly scattered across the graph with none of them obviously standing out in terms of the number of edges. The hierarchy structure tells us that there are a few very important people that participate in most of the issues but since there is only a small number of people in the middle of the hierarchy, the hierarchy is not very well balanced.

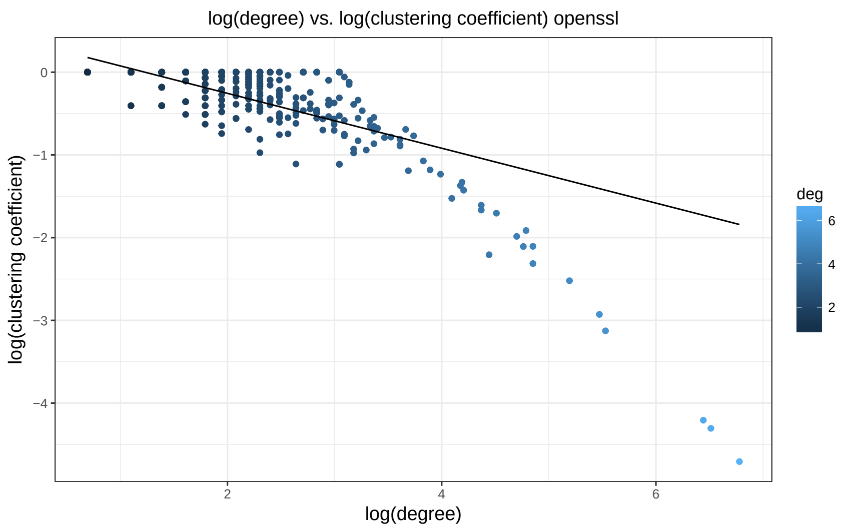


Figure 4.1: Hierarchy Graph for the OpenSSL issue-based author network

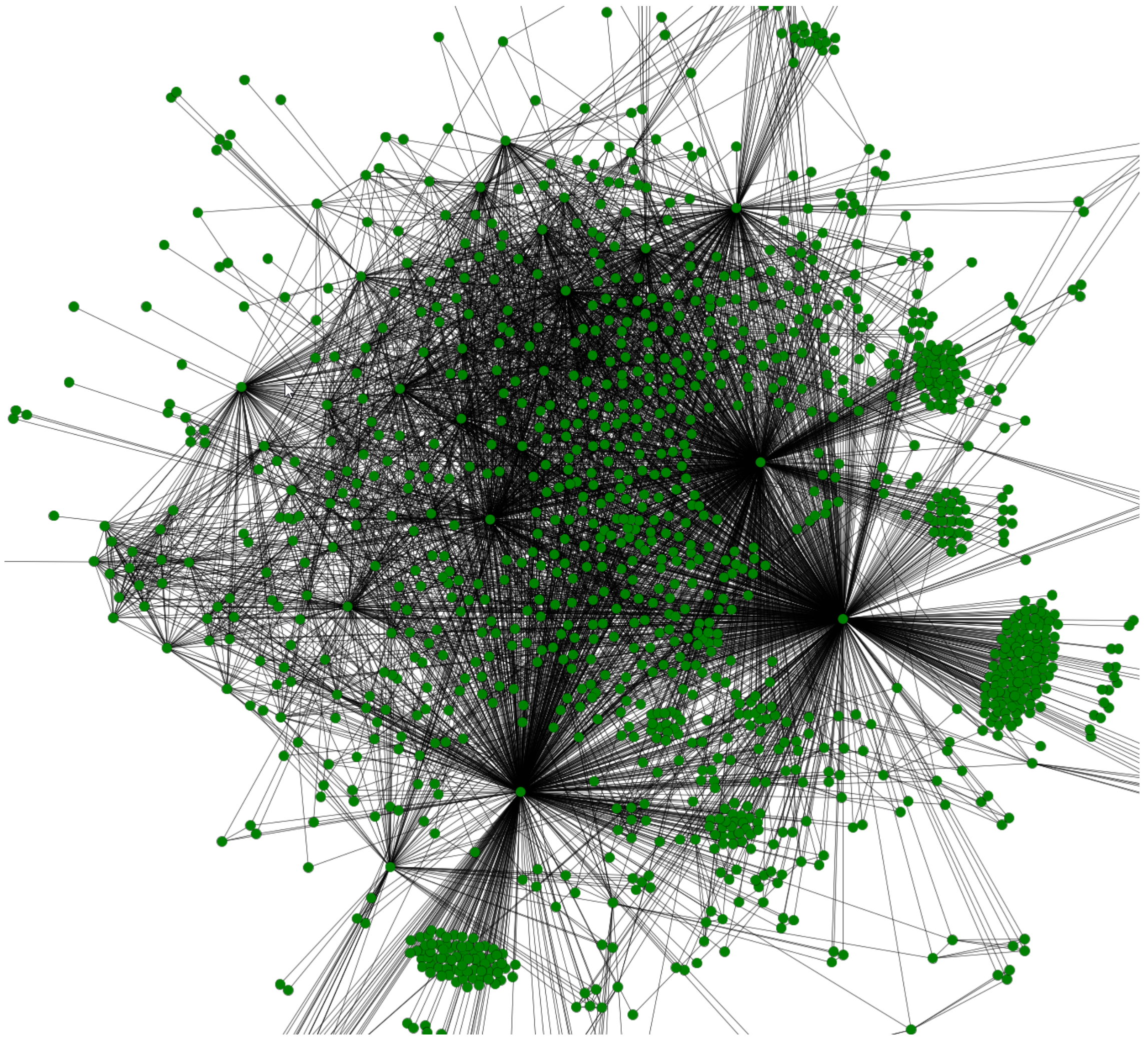


Figure 4.2: Graph visualization for issue-based author network for OpenSSL. The nodes represent people who have contributed to the issues in the project, an edge between two nodes means that the two issue authors have participated in the same issue.

Comparison of Issue-Based and Mail-Based Author Networks

We now take a look at the mail-based author network for OpenSSL and check if the metrics show a different outcome or whether we get similar results. Since the mailing-list contains data from 2001 to 2016 while the issues only begin in 2013, the mail-network is significantly larger than the issue-based one both in the number of nodes and the number of edges. In order to get a comparable amount of data, we cut the earlier years of the mailing-list so the number of edges is about equal. We consider the number of edges more important than the number of nodes because the interactions are more interesting to us than the number of participants.

Simple Metric	Value (Mail)	Value (Issue)
Number of Nodes	1533	1354
Avg. Degree	7.14	7.926
Avg. Pathlength	3.028	2.284
Max.Total Degree	408	880
Density	0.00466	0.00586
Complex,Metrics		
Clustering,Coefficient (L)	0.729	0.919
Clustering,Coefficient (G)	0.137	0.0484
Modularity	0.342	0.137
Smallworldness	38.182	15.074
Scale-Freeness		
alpha	1.89	2.397
x min	3	6
KS p	0.152	0.879

Table 4.3: Comparison between mail-based and issue-based author network for OpenSSL

In Table 4.3 we can see that the number of nodes for the mail network is about 200 higher, combined with the average node degree, which is a bit lower, the number of edges is about equal to the issue-based network. The density is a little lower and the average path length is a bit higher. The only simple metric that shows a clear difference is the maximum total degree. It is only half of what it is in the issue based network. This is also reflected in Figure 4.4 where we can observe nodes with more connections than the rest but not as distinctly as in Figure 4.2.

The clustering coefficient shows roughly the same characteristics as it does for the issue-based network. The average local clustering coefficient (marked with (L)) is fairly high, once again most likely caused by a lot of low degree nodes. The global clustering coefficient (marked with G) is low again even though it is twice as high as it was for the issue-based network which is a noticeable difference, so the mail-based network seems to show a stronger clustering behavior.
The modularity is considerably higher for the mail-based network than it is for the issue-based one. Since the value is over 0.3, we consider it to be a modular network. The reason might be that people keep more to mail threads that they are strictly involved in while in issues people respond more liberally if they have something to contribute and thus cause the network to have less well defined modules. However

this is speculation since we only compare the network metrics for mail-based and issue-based author networks for one project.
The mail-based author network, like the issue network is scale-free. The p-value is lower, but it is still over 0.05. Xmin is also lower which means that more of the nodes fit with the power-law's degree distribution. With a value of about 38, the small-worldness is way beyond one, so the mail-based author network for OpenSSL is a small-world network, just like with the issue-based one. Since, as mentioned above, most social networks are small-world networks, this is not surprising.
Lastly, we take a look at the hierarchy graph for the mail network, depicted in Figure 4.3. While the base structure is similar once again, the nodes in the mail-based network are more evenly spread out. There are more nodes with higher degrees and a lower clustering coefficients but at the same time, the maximum hierarchy position is not as high as the highest ranked nodes issue-based network. Altogether there is a stronger, more obvious hierarchy in the mail-based network.

Conclusively we can say, that issue-based and mail-based author networks have similar tendencies as far as the metrics that we apply are concerned. The results differ in some aspects, but they both share the same tendencies, which makes sense considering both of them are based on communication in a software development environment and the data sources fulfil a similar purpose.

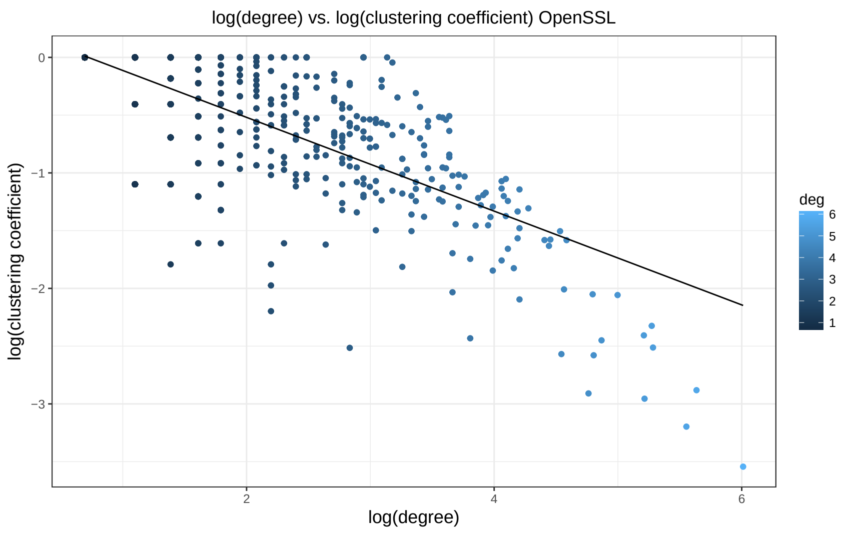


Figure 4.3: Hierarchy Graph for the OpenSSL mail-based author network

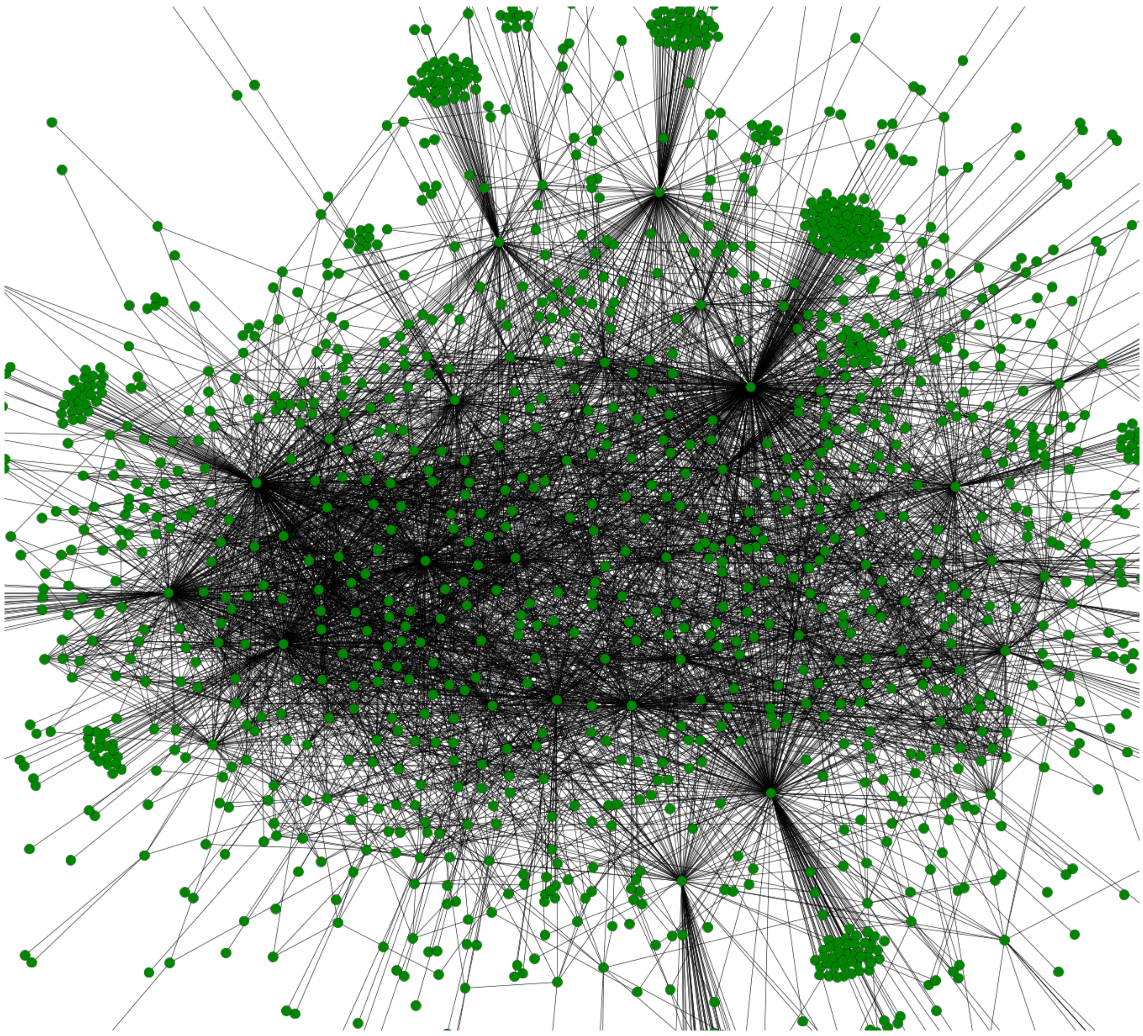


Figure 4.4: Graph visualization for the mail-based author network for OpenSSL. The nodes represent people who have written mails in the mailing-list, an edge between two nodes means that the people have written mail(s) in the same thread.

4.2.2 Other Projects

After analyzing the issue-based author network for OpenSSL and comparing it to the mail-based network, we now take a look at the issues-based networks of several other projects.

Once again, we first look at the simple metrics shown in Figure 4.5 and cover the more complex metrics afterwards. The number of nodes and maximum degree show the relative scale of the issue usage of the projects. Owncloud is by far the largest project in terms of the number of authors and the maximum degree, which fits in with the number of issues. Interestingly, the average degree in general does not seem to increase with the number of issues, though we cannot make a universal claim since we only look at five projects. The average path length appears to be independent from the size of the network. Density on the other hand seems to decrease with the number of nodes which indicates that the number of connections does increases fairly slow compared to the number of nodes.

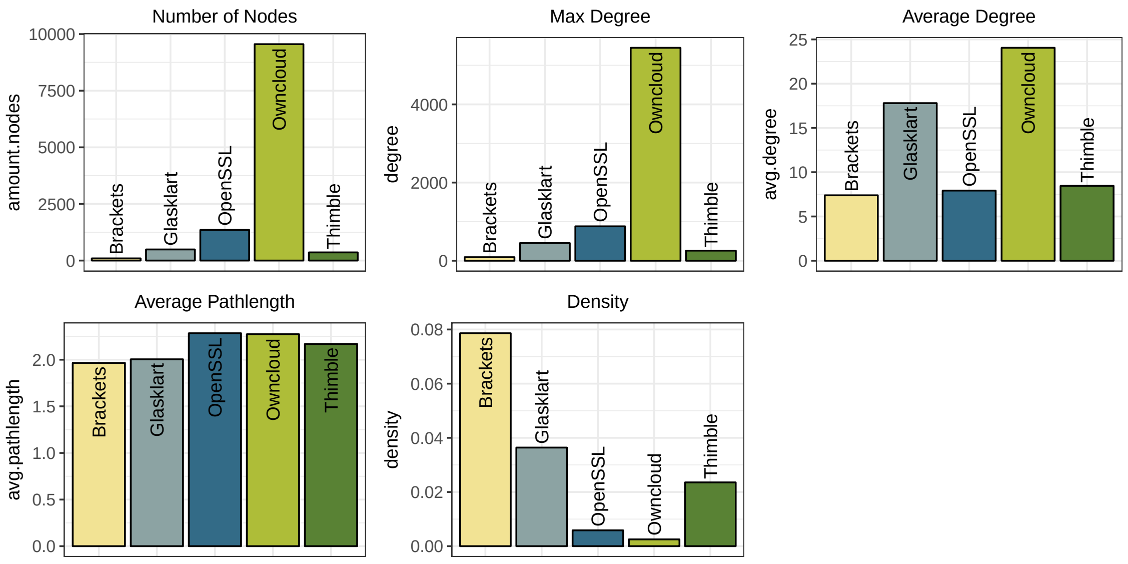


Figure 4.5: Results of the simple metrics applied to the issue-based author networks of the five projects: Brackets, Glasklart, OpenSSL, Owncloud and Thimble.

Now, we examine the remaining, more complex metrics that we have depicted in Figure 4.6 and Figure 4.7. The local clustering coefficient (marked with (L)) is very high and almost equal in all of the networks. This is most likely due to a lot of nodes with few edges in all of the networks, have too large of an impact on the result. The global clustering coefficient (marked with (G)) shows more interesting results. We can observe a quite strong divergence in the global clustering coefficient that does not seem to correspond with the network size. Since there is no obvious reason for the different clustering coefficients, further research would have to be done to determine the cause of this.
The modularity value seems to be about equal for most of the network with the exception of Owncloud, which still does not go above the value of 0.3 to be considered a highly modular network. The reason for Owncloud being having a higher

modularity is not evident from the data and results we have. It could be connected to size since OpenSSL, the second largest project in terms of issue data, also shows a slightly higher modularity but this is not conclusive, especially since the rest of the projects, despite their size differences, show no clear difference in modularity.
From the data that we have, the small-worldness value of the issue-based networks appears to scale with the size of the corresponding network, though they are all categorized as small-world networks since the value for smallest network, Brackets, is about 2.2.
All of the five networks, except Owncloud, are considered scale-free since the p-value after fitting a power law is above 0.05 for all of them. For the four projects that are scale-free, this means that they have a few central hub nodes with high degrees and plenty of lower degree nodes. Since the network for Owncloud, like the others, has a high maximum node degree and a comparatively low average node degree, something must be different with the curve of the degree distribution. However we do not know why this is the case or what causes it.

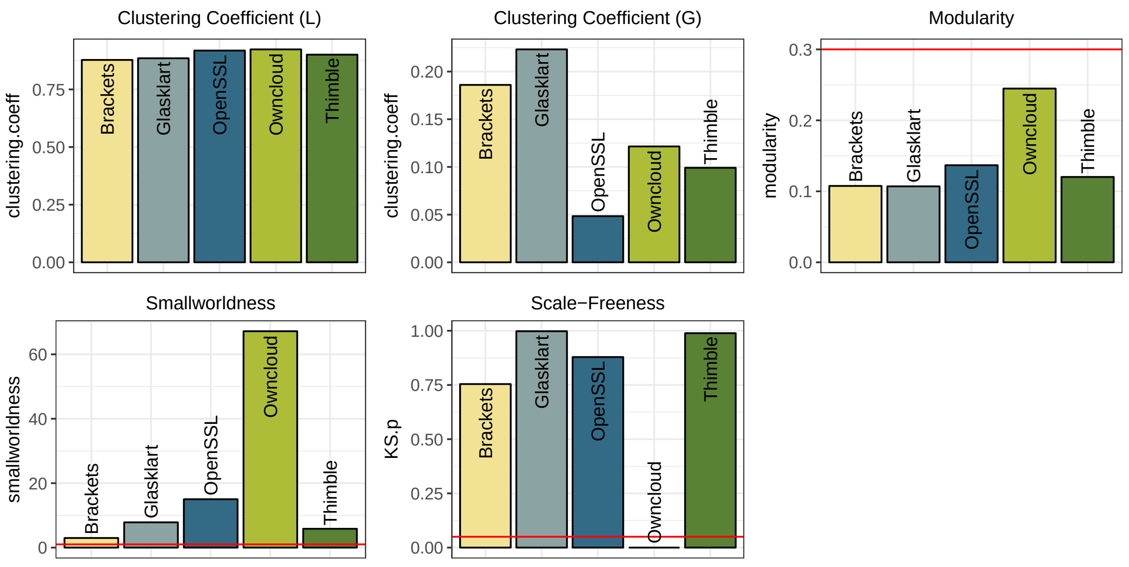


Figure 4.6: Results of the complex metrics applied to the issue-based author networks of the five projects: Brackets, Glasklart, OpenSSL, Owncloud and Thimble. The red lines in the plots for modularity, small-worldness and scale-freeness show the minimum value for the network to be considered modular, small-world and scale-free.

Finally we inspect the hierarchy results in Figure 4.7. They all show very similar hierarchy characteristics. All of them seem to have a hierarchy with few nodes with drastically higher degrees and lower local clustering coefficients and most of the nodes having low degrees and high local clustering coefficients. The hierarchy behavior seems to be independent of network's size.

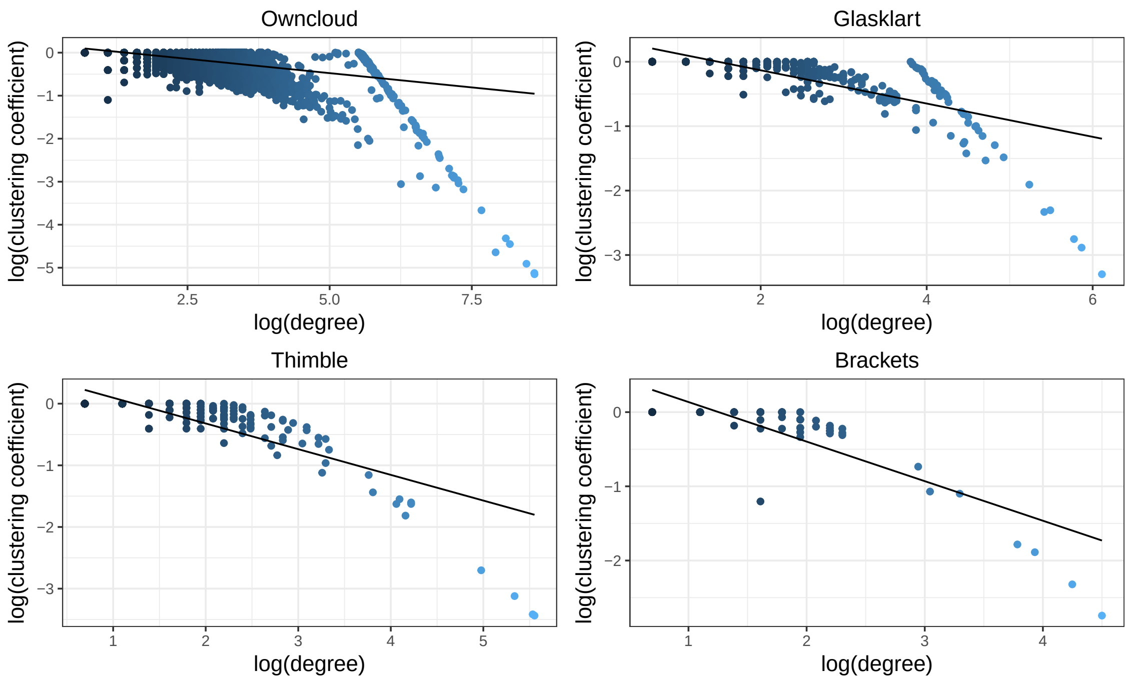


Figure 4.7: Hierarchy plots for the projects except OpenSSL. The hierarchy plot for OpenSSL can be found in Figure 4.1

5 Conclusion

Because communication is an essential part of software development, it is important to get a better understanding of how this communication works and how it influences the development process itself. Up to this point, plenty of research has been done using mailing-lists as the source of communication data but an increasing portion of the communication around software projects is conducted using other means than e-mails like the GitHub issue tracker that we use as our source for communication data. By analyzing this issue data, we hope to provide a complementing in addition to the mail data, as well as a replacement option for projects that do not use a mailing-list. Since issues are a way of communication that is integrated into the development environment, they might also open up new ways of connecting communication to the collaboration on the source-code.

To determine the usefulness of the issue data, we have integrated it into a tool-chain for network-based software project analysis. We first mined the data from REST API that GitHub offers. We then prepared the issues for network construction and matched it to the data of that is already in use, so both types can be used together in later research. From the prepared issue data we then constructed issue-based author networks that we then used to perform some metric-based network analyses. To get an estimate for how issue and mails compare as a communication medium, we constructed networks for both and compared the metric results. Even though no clear definite conclusion is possible, since we only applied the comparison of mail-based and issue-based networks to OpenSSL, this case-study shows a lot of similarities among the two data sets. Modularity and the maximum total degree showed the strongest divergences where the mail-based network is categorized as strongly modular, while the issue-based network is not and the issue network has over twice the maximum total degree as the mail-based one. As for the comparison of the five projects that we analyzed the issue-based networks for, OpenSSL, Glasklart, Thimble, Brackets and Owncloud, the most surprising result was the differences in the clustering coefficient and that Owncloud's issue-based author network was not scale-free as compared to the other four networks. The clustering coefficient was varying considerably between the different projects and it did not appear to correlate to the size of the network. We cannot explain the differences in the clustering coefficient or the sticking out of OWNCLOUD in terms of scale-freeness with the analyses we performed but they could be a candidate for future research on issue-based author networks.

In conclusion, the issue-based networks seem to show fairly consistent results to the point where they could be used as either an addition or as a replacement for mailing-lists in the network based analysis of software projects. Still because of the small sample size that we used for our analyses, more research should be done to ensure the consistency over a larger number of projects. For the future, utilizing issue-based networks for analyzing a project's development over time, as well as researching topics that have already been analyzed with mail-based networks, might deliver interesting results.

Bibliography

[AJS11] A. Sillitti A. Jermakovics and G. Succi. Mining and visualizing developer networks from version control systems. *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, page 24–31, 2011. (cited on Page 3 and 4)

[BA99] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 1999. (cited on Page 6)

[BE05] Ulrik Brandes and Thomas Erlebach. *Network Analysis: Methodological Foundations.* Springer, 2005. (cited on Page 4 and 5)

[BPD$^+$08] Christian Bird, David Pattison, Raissa D'Souza, Vladimir Filkov, and Premkumar Devanbu. Latent social structure in open source projects. *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 24–35, 2008. (cited on Page 4)

[Cha05] Robert N. Charette. Why software fails. *IEEE Spectrum*, pages 42–49, 2005. (cited on Page 1)

[DN10] Arne Beckhaus ans Lars M. Karg Dirk Neumann. The impact of collaboration network structure on issue tracking's process efficiency at a large business software vendor. *Proceedings of the 43rd Hawaii International Conference on System Sciences*, 2010. (cited on Page 9)

[HG] Mark D. Humphries and Kevin Gurney. Network 'small-world-ness': A quantitative method for determining canonical network equivalence. (cited on Page 5)

[HM01] J. D. Herbsleb and D. Moitra. *Global software development*, volume 18. IEEE, Mar 2001. (cited on Page 1)

[Job17] Mitchell Joblin. *Structural and Evolutionary Analysis of Developer Networks.* PhD thesis, Universität Passau, Germany, 2017. (cited on Page 4 and 8)

[LLFGB04] G. Robles L. Lopez-Fernandez and J. M. Gonzalez-Barahona. Applying social network analysis to the information in cvs repositories. *1st International Workshop on Mining Software Repositories (MSR)*, page 101–105, 2004. (cited on Page 3 and 4)

[LLH06] Jesus L. Lopez, G. Robles and I. Herraiz. Applying social network analysis techniques to community-driven libre software projects. *International Journal of Information Technology and Web Engineering*, page 27–48, 2006. (cited on Page 3 and 4)

[MJM17] Claus Hunsen Mitchell Joblin, Sven Apel and Wolfgang Maurer. Classifying developers into core and peripheral: An empirical study on count and network metrics. *IEEE/ACM 39th International Conference on Software Engineering*, 2017. (cited on Page 8)

[NG] Mark E.J. Newman and Michelle Girvan. (cited on Page 6)

[RR14] Mohammad M. Rahman and Chanchal K. Roy. An insight into the pull requests of github. 2014. (cited on Page 9)

[WF95] Stanley Wasserman and Katherine Faust. *Social Network Analysis: Methods and Applications.* Cambridge University Press, 1995. (cited on Page 4)

[WS98] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 1998. (cited on Page 5)

[XF14] Qi Xuan and Vladimir Filkov. Building it together: Synchronous development in oss. *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 222–233, 2014. (cited on Page 4)

Eidesstattliche Erklärung:

Hiermit versichere ich an Eides statt, dass ich diese Bachelorarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Raphael Nömmer

Passau, den 30. September 2017