University of Passau

Department of Informatics and Mathematics

Bachelor's Thesis

Using Multiplex Centrality Measures to Identify Core Developers in Open-Source Software Projects

Author:

Anselm Fehnker

September 21, 2019

Advisors:

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

Thomas Bock
Chair of Software Engineering I

Angelika Schmid
Junior Research Group PICCARD

Barbara Eckl-Ganser
Junior Research Group PICCARD

Fehnker, Anselm:
Using Multiplex Centrality Measures to Identify Core Developers in Open-Source Software Projects
Bachelor's Thesis, University of Passau, 2019.

Abstract

The identification of core developers in *Open-Source Software* (OSS) projects is an important but challenging topic. It is essential for the understanding and the improvement of the OSS development process to know which developer is a core developer. A common approach to identify core developers is to transform OSS projects into socio-technical developer networks in which central nodes can be identified by centrality measures. This central nodes give information about which developers are the most important developers in the OSS project. Because the established centrality measures are not able to consider dependencies between different types of interactions in OSS projects, we introduce a new approach to apply multiplex centrality measures on OSS projects. In particular we apply a multiplex centrality measurement called *Essential nodes Determining based on CP Tensor Decomposition* (EDCPTD).

Therefore OSS projects are modeled as forth-order tensors. The decomposition of this tensor shall provide information about the importance of developers in OSS projects. This multiplex centrality measurement has not been applied on OSS projects yet. To evaluate if this multiplex centrality measurement is useful for the identification of core developers in OSS projects and to examine if the multiplex approach has advantages over the established centrality measures, we implement EDCPTD centrality for OSS projects and conduct different research questions. These research questions analyze how EDCPTD centrality behaves compared to the already established centrality measures degree- and eigenvector centrality and how EDCPTD centrality is able to identify cor developers on the basis of their behavior. The results of these research question demonstrate, that EDCPTD centrality is in the most cases able to identify core developers in OSS projects.

Contents

1 Introduction

1.1 Goal of this Thesis

Many popular software projects such as Mozilla Firefox, Google Chromium, Android or Libre Office have been developed as Open-Source Software (OSS) projects. This kind of software projects have public source codes for everybody to read, use and work on. Therefore volunteer software developers from all over the world can contribute to them. In many cases these developers are interested users of the software that have different motivations for their work [FG07]. So it is not unusual that developers with different interests and different skills work together on the same code. Further it is not unusual and that these developers do not know each other personally. Because of this piece of setting, problems that are in the clear hierarchically structured commercial software development unusual, can occur more likely in the development of OSS projects. For example developers can have degenerated discussions on how things should be implemented, several developers can have implemented the same functionality due to missing agreement, or developers with insufficient programming knowledge can contribute to a project.

To ensure that the software development process in OSS projects proceeds in an ordered way, many interactions between the developers are necessary. New tasks have to be distributed to appropriate developers, new code contributions have to be checked if they are flawless and consistent to the other code in the project or already existing code has to be maintained as it could get outdated. To process all these interactions in a regulated manner it is essential that the large groups of volunteer software developers are organized in a hierarchical structure [QHB11]. As the hierarchy of software developers in OSS projects is not ordered in progressive stages as in commercial software development, OSS developers are often divided into different groups of core and the peripheral developers [CS17]. Core developers are developers who are a permanent part of the project. Beside their code contributions, they also play an important role in the organization of the project. For example, they distribute tasks to other developers, check if new code contributions fit to the project, keep an eye on the maintenance of the source code or help other developers in the community. Therefore they have many permissions in the OSS project [KCH06]. In contrast, peripheral developers are developers who only temporarily contribute code to the project. They usually do not play an important role in the organization of the project and therefore do not get more permissions than necessary [PSC12]. For example, code of peripheral developers is usually checked by core developers first before it becomes part of the OSS project.

To improve the organization and the maintenance of OSS projects and to reduce costs it is important to understand which developer is a core developer. Therefore it would be useful to figure out the importance of the developers in the project with the help of measurable data. A common approach to rate how important developers in OSS projects are, is to transform the OSS project data to a developer network and apply mathematical centrality measures on it. This will return information about the centrality of the developers in the network and thereby also about the importance of the developers in the OSS project.

As the for the identification of core developers established centrality measures do not consider all relationships from OSS projects for their calculations, we will examine a new approach to identify core developers in OSS projects. Therefore we will introduce a new way to represent OSS projects as multiplex networks and apply a multiplex centrality measurement called EDCPTD centrality on them. This multiplex centrality measurement is based on tensor decomposition and has not been used to identify core developers in OSS projects yet.

The main research question of this bachelor thesis is, weather this new representation of OSS projects and the application of the new multiplex centrality measurement EDCPTD centrality can be used to identify core developers.

1.2 Structure of the Thesis

First in Chapter 2 we will provide background information and explain what OSS projects are, how OSS development works and how OSS developers can be divided in a hierarchical core and peripheral structure. Here we will especially point out what typical patterns of behavior for core developers in OSS projects are.

After this, in Chapter 3, we will declare how OSS projects can be modeled as networks and introduce the centrality measures degree- and eigenvector centrality that are able to identify core developers in these network representations of OSS projects. Further we will introduce the multiplex centrality measurement EDCPTD centrality that is currently used to identify central nodes in real-world biological networks but might also be interesting to identify core developers in OSS projects.

To examine if EDCPTD centrality is able to identify core developers in OSS projects, we present two research questions in Chapter 4. How this research is arranged is explained in Chapter 5. Especially the transformation of OSS data to multiplex networks, the implementation of EDCPTD centrality, as well as the execution of the research questions are described.

The evaluation of the research questions will be presented in Chapter 6. After this we will give a conclusion how EDCPTD centrality is able to identify core developers in

OSS projects in Chapter 7. At the end in Chapter 8, a perspective how the research questions can be extended to gain even more information about the application of EDCPTD centrality is given.

2 Background

In this chapter an introduction about OSS projects and how they are organized is given. Further it is explained which tasks core developers have in OSS projects and how they distinguish from peripheral developers.

2.1 OSS Projects in General

Open-Source Software (OSS) projects are by definition software projects in which users have free access to the source code. In these projects is the software development usually not centrally planed but a decentralized self-organized process [GM02].

Nevertheless big software projects have been developed this way. These include prominent projects like the Internet browsers MOZILLA FIREFOX and GOOGLE CHROMIUM, the operating systems LINUX and ANDROID, the programming languages PERL and PYTHON, and much more. For the development, the enhancement and the maintenance of these large OSS projects, a huge number of developers is needed. The developers are mostly voluntary users of the developed software. The motivation for their work can have different reasons. Some want to signal their quality or improve their future job prospects, others just contribute for fun or for the reputation to be part of the project [FG07] [Kri06]. This has the consequence that developers from all over the world who do not know each other personally and who can have different programming competences work on the same source code.

To avoid misunderstandings and to ensure a regulated flow of work, a lot of organization and communication is necessary. To simplify this process, OSS projects usually provide mailing lists or supporting platforms such as GITHUB[1], GITLAB[2], or JIRA[3].

A mailing list is a simple tool that helps OSS developers to communicate with each other. This includes discussions about how things should be implemented or the

[1]https://github.com/(Accessed at 20.09.2019)
[2]https://about.gitlab.com/(Accessed at 20.09.2019)
[3]https://www.atlassian.com/de/software/jira(Accessed at 20.09.2019)

assignments of tasks. The typical use of mailing lists is that one developer writes a message in which he asks for help, suggests improvements, presents new ideas, or starts a discussion. Other interested developer start replying to this initial mail. This group of messages is called a mail thread. With the help of threads developers can coordinate their work and the organization of the project stays documented. This makes it easy for new developers to retrace decisions. [SLT09]

Similar advantages in the organization and the documentation of the project can be achieved by the usage of platforms such as GitHub[4], GitLab[5], or JIRA[6]. All three platforms support a concept called issues. Likewise to the start of a mail thread, a developer can open an issue about problems in the project or possible improvements. Underneath this issue other developers can write comments to discuss solutions or to share their opinions. Further these issues provide the opportunity to directly assign developers that are responsible for its solution. After the issue has been processed it can be marked as resolved in order to notify developers that this problem or discussion is no longer active. So again a comprehensible flow of work is ensured.

In addition GitHub and GitLab provide even more advantages in the assistance of the OSS development process as they do not only simplify the communication and organization of the project but also have a direct interface to the source code. As the name already suggests they are based on the version control system GIT[7] and can be used as a code hosting repository. This makes it easier and safer for developers to contribute to the project. Instead of directly changing the source code of the project, developers can clone the source code into private forks and apply changes there. This way the original source code stays unchanged while the developers work on their private copy. Thereby several developers can make changes on the same code file, without interfering each other. After a developer has completed his changes, he can create a pull-request. This pull-request is kind of an application for the changes to become part of the actual project. Pull-requests are internally handled as issues, which means that developers can write comments underneath the pull-request and give feedback or request changes. When the new code fulfills all demands it can directly be merged to the original project with the help of GIT. This procedure is especially reasonable for OSS projects as it helps to prevent that unqualified developer contribute to the project.

2.2 The Role of Core Developers in OSS Projects

In fact, OSS developer collaborate and communicate in many different ways with each other. This different types of interactions between developers makes OSS development a complex process. In order to keep this process regulated and enable a high quality software development, a hierarchical structure is needed [Abe07]. But how can volunteer developers be divided in such a hierarchical structure? A common approach is to classify OSS developers in an onion model [CH03]. This model was originally invented by Nakakoji et al. ([NYN+02]) with eight different types of project members. Nowadays the onion model is often reduced to four different types

[4]https://github.com/(Accessed at 20.09.2019)
[5]https://about.gitlab.com/(Accessed at 20.09.2019)
[6]https://www.atlassian.com/de/software/jira(Accessed at 20.09.2019)
[7]https://git-scm.com/(Accessed at 20.09.2019)

of project members Figure 2.1 [CH03] [Abe07]. This reduced form is sufficient to understand the hierarchy for our application.

In the middle of the onion are the core developers. Core developers are the group of developers that usually contribute most of the code and control the design and evolution of the project. Around the middle are different layers of peripheral developers. The first layer around the core developers consist of co-developers. These are developers that contribute temporary to the project. This can include submitting bug fixes or modifying already existing code. In the most projects a core developer must review the contributions of co-developers before they get merged to the OSS project. The group of co-developers is usually much larger than the group of core developers. Around the co-developers is the layer of active users. This consist of users that do not directly contribute to the project but affect the development for example by reporting bugs. On the very outside of the onion-model are the passive users, that only use the software developed by the OSS project but do not contribute to it [CH03] [CS17].

In this thesis we engage with the question how core developers can be identified in this structure. Therefore a closer look on the pattern of behavior of core developers in OSS projects is necessary. In particular a delimitation between the behavior of core and co-developers is wanted.

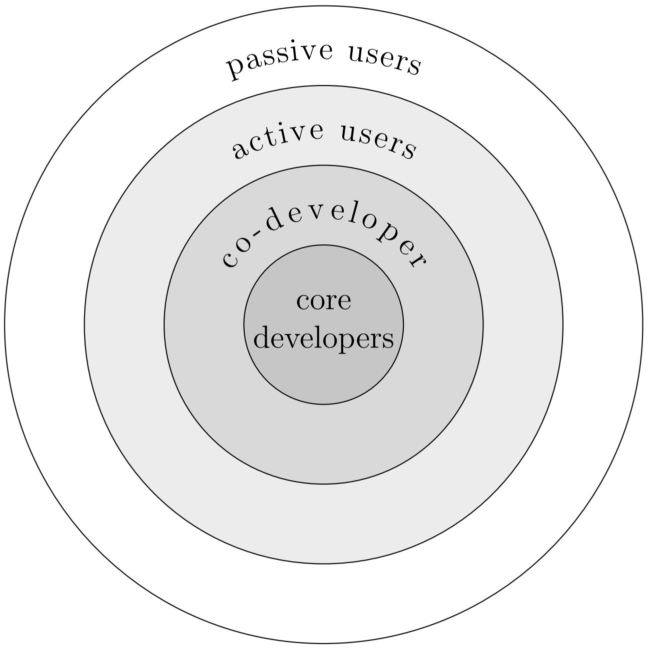

Figure 2.1: An onion model of OSS development. In the middle are the core developers. Around the middle are several layers of peripheral developers.

A main difference between core and co-developers is that core developers play an important role in the leadership of the project. They usually have a detailed knowledge of the system architecture and often feel strongly associated to the part of the project and the code files that they manage [JAHM17]. All this is reflected in their behavior and activity on the OSS project. Because of their role in the leadership and organization in the OSS project, core developers tend to be more active in the communication than peripheral developers [CS17]. This includes extraordinary much communication between core developers [JAHM17] and a high integration in the communication with peripheral developers. So peripheral developers do not often directly communicate with each other but have a core developer as middlemen [CS17]. But not only the frequent and significant communication is striking for core developers. Further most code contributions with new functionalities are made by core developers while peripheral developers more often only modify code. Some studies found that in some projects 80% of the code is written by just a small group of developers [CH03].

After all, the behavior of core and peripheral developers differs in several ways. Core developers have a more important role in collaboration and communication than peripheral developers. This provides evidence to identify core developers in OSS projects. A method that rates the importance of developers in OSS projects would suggest which developers are adequate core developers. Therefore OSS projects are often considered as social networks such that centrality measures can be applied on them. The idea behind a centrality measurement is to calculate a score that represents the centrality of a subject in a social network and thereby gain information about the importance of the corresponding participant in the real compound. [PG06].

Because OSS development consists of much more than just programming, all different aspects of OSS development have to be considered in the centrality measurement. As well the code contributions as the activities on a mailing list or on issues have to be considered in the identification of core developers. So adequate centrality measures for OSS projects have to be able to handle different types of interactions in social networks.

3 Network Modeling and Centrality Measurement for OSS Projects

In this chapter established approaches to model OSS projects as networks and identify core developers with one-dimensional centrality measures are introduced. As this approaches are not able to consider the influence that different types of interactions in the OSS project have on each other, we further introduce a multiplex centrality measurement called EDCPTD centrality, from which we expect to consider this influence. This multiplex centrality measurement is currently primarily used for other areas of research, but could be adapted for the use on OSS projects.

3.1 OSS Projects as Developer Networks

Centrality Measurements which are commonly used to identify core developers in OSS projects are degree centrality and eigenvector centrality. As recent studies by Joblin et al. ([JAHM17]) have shown, do these one-dimensional centrality measurements reflect the roles of developers in OSS projects accurately. Both centrality measures are based on socio-technical developer networks. This is a kind of social networks, in which developers are represented by nodes and interactions between developers are represented by connections between the corresponding nodes [JAM17]. This developer networks can be represented by a graph (Figure 3.1).

Definition 3.1.1. *In a mathematician's terminology, a **graph** is a collection of points and lines connecting some (possibly empty) subset of them. The points of a graph are most commonly known as graph vertices, but may also be called "nodes" or simply "points". Similarly, the lines connecting the vertices of a graph are most commonly known as graph edges, but may also be called "arcs" or "lines". [Weia]*

In order to transform the data of OSS projects to a developer network, a breakdown to the specific interactions between OSS developers is necessary. For every type of

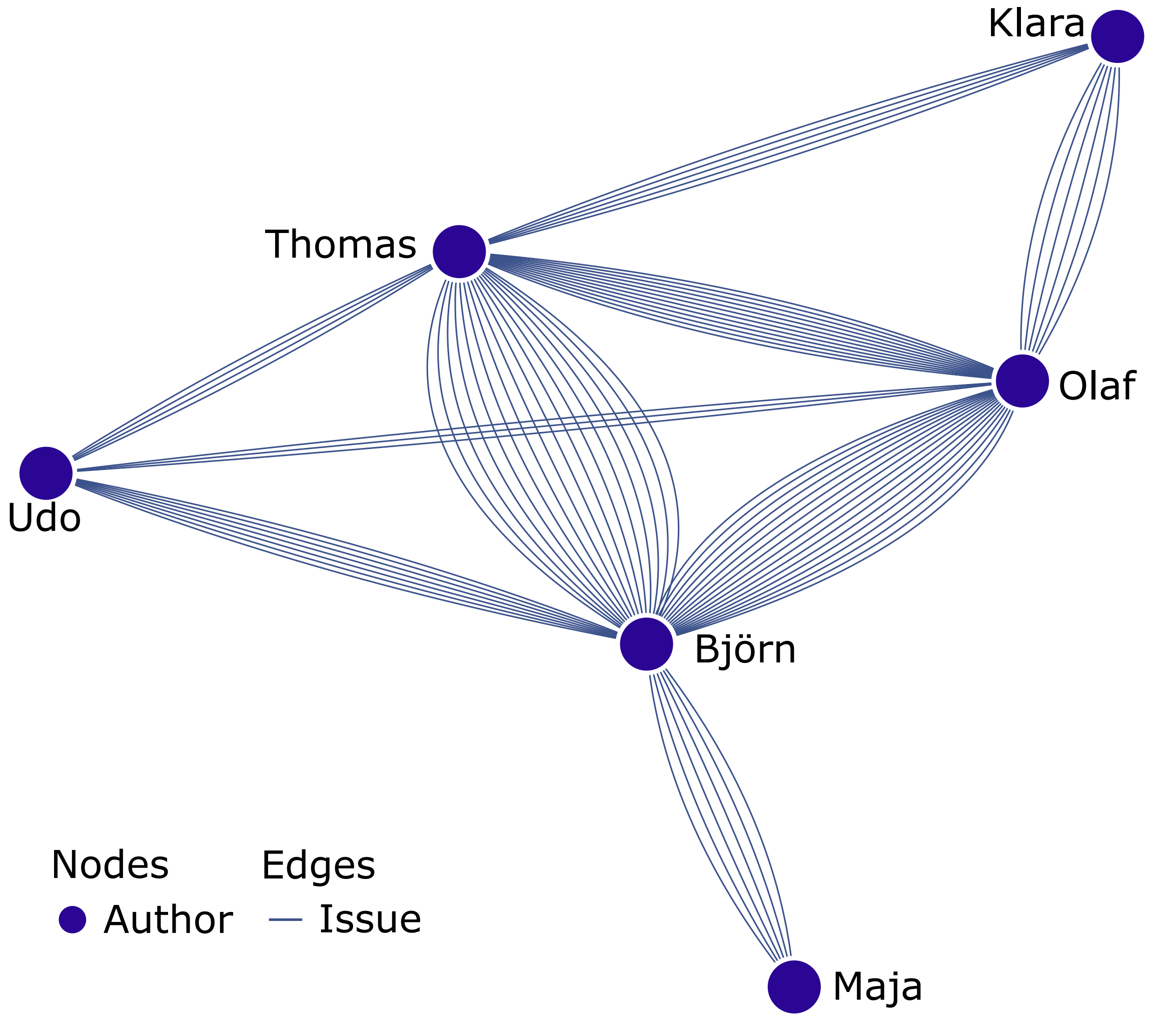

Figure 3.1: A graph representation of the developer network that models the issue interactions between developers in an example project. A connection between two nodes means that the two developers have been active on the same issue. This is an undirected and weighted graph. In this representation the weights of the edges are displayed by several connections between two nodes. This graph has been plotted with CORONET

interaction a different developer network is build. When a developer is active for the first time in an OSS project, a node that represents his activity is added to the developer networks. When two developers first interact with each other in one type of interaction, an edge between the nodes that represents the two developers is added to the corresponding developer network. Every time these two developers interact with each other in this way again, the weight of this edge is increased.

In our application we concentrate on three different types of interactions: First we consider the code contributions that developers make on the same code files to program the project. Further we examine the two ways of communication that where introduced earlier, the mails developers write via the mailing list to each other and the issues developers are active on together. So three different developer networks are build, the cochange network that represents if developers have made changes on the same code files, the mail network that represents if developers have communicated via the mailing list with each other and the issue network that represents if developers have been active on the same issues. All of these networks have the same amount of nodes, the developers that work in the OSS project. The edges between the developers in these networks represent if and how often the developers have interacted in this type of interaction with each other. For example if two developers contribute to the same code file, an edge between their corresponding nodes on the cochange network is added or, if already existing, the weight of the edge between them in the cochange network is increased. Analogous if a developer writes an email to another developer the weight of the edge between them in the mail network increases and if two developers are active on the same issue the weight between them in the issue network increases. So the developer networks only contain connections with weights that are integers and bigger than zero. Note that the graph representation of developer networks can either be weighted or unweighted and directed or undirected.

Definition 3.1.2. *A **weighted graph** is a graph in which each edge is given a numerical weight. A weighted graph is therefore a special type of labeled graph in which the labels are numbers (which are usually taken to be positive). An **unweighted graph** is a graph in which edges have no weights. [Weie]*

Definition 3.1.3. *A **undirected graph** is a graph for which the relations between pairs of vertices are symmetric, so that each edge has no directional character (as opposed to a **directed graph**). [Weid]*

However, the centrality measurements that are used to identify core developers in OSS projects, cannot directly be applied on a graph representation of a developer network like in Figure 3.1. The common representation with nodes and edges of such a graph might be a good visualization for the human eye to recognize relations between nodes, but nevertheless it is not adequate for centrality calculations. To apply centrality measures on this developer networks, the adjacency matrices that represent the connections in the graphs are needed (Figure 3.2). These adjacency matrices contain exactly the same information as the graph representation but in a mathematically more useful way.

$$A = \begin{bmatrix} 0 & 0 & 6 & 17 & 16 & 7 \\ 0 & 0 & 0 & 6 & 5 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 \\ 17 & 6 & 0 & 0 & 12 & 3 \\ 16 & 5 & 0 & 12 & 0 & 4 \\ 7 & 0 & 0 & 3 & 4 & 0 \end{bmatrix}$$

Figure 3.2: The corresponding adjacency matrix to the graph in Figure Figure 3.1. The authors are ordered by alphabet: Björn, Klara, Maja, Olaf, Thomas, Udo

Definition 3.1.4. *For a graph with n nodes the **adjacency matrix** $A \in \mathbb{R}^{n \times n}$ is defined as follows:*

$$(A)_{ij} = \begin{cases} w_{ij} & \textit{if node } i \textit{ points to node } j \textit{ with weight } w_{ij} \\ 0 & \textit{otherwise} \end{cases} \tag{3.1}$$

If the graph is unweighted w_{ij} is 1 if the nodes i and j are connected and 0 if not. [Lui13]

As the graphs had only positive weighted edges the corresponding adjacency matrices that represent the software projects have also only nonnegative entries. With this adjacency matrices it is possible to identify core developers in OSS projects using degree- and eigenvector centrality.

3.2 Established Centrality Measures for OSS Projects

The one-dimensional centrality measures degree- and eigenvector centrality are confirmed able to determine developer roles in OSS projects [JAHM17]. In the following section, is explained how these two centrality measures work.

3.2.1 Degree Centrality

Degree Centrality is a centrality measurement that is based on the number of adjacencies a node has in the network, i.e. the number of connections to other nodes a node has [TOS10]. For a node i in a given network with adjacency matrix $A \in \mathbb{R}^{n \times n}$ the degree centrality can be calculated by summing up all weights of edges that point to the node.

Definition 3.2.1. *For a node i in a network with n nodes the degree centrality is defined as:*

$$DC_i = \sum_{j=1}^{n} a_{ji} \tag{3.2}$$

Where a_{ji} is a entry in the adjacency matrix A of the network. [TOS10]

The node with the highest degree centrality on an adjacency matrix is the most important one in the corresponding developer network. Therefore the developers, which nodes are rated high by degree centrality in all three developer networks, are identified as core developers. As the calculation of degree centrality only adds entries of the adjacency matrix it is a centrality measurement that is quite easy to calculate. But in the calculation of this centrality measurement, only direct connections between two developers are considered. It does not make a difference if two developers are completely independent of each other or an indirect connection, for example via a third developer that often interacts with both of them, exists. In a real OSS project this makes a difference.

Another disadvantage of the centrality measurement is that all edges are considered with the same importance. This can lead to mistakes in the centrality measurement, as some edges, for example edges that link different groups of developers should be considered as more important than other edges. In conclusion degree centrality only gives a quick overview about who are the most active developers in this interaction.

3.2.2 Eigenvector Centrality

Eigenvector Centrality is in opposite to this a centrality measurement that is able to consider these indirect relations and the importance of connections in the network [Bon07]. To understand the functioning of eigenvector centrality, it is necessary to look a little bit deeper into mathematics, especially in linear algebra. First it is essential to know what an eigenvector is and when it exists.

Definition 3.2.2. *Let F be an endomorphism of the K-vectorspace V. A $\lambda \in K$ is called **eigenvalue** of F, if a $v \in V$ with $v \neq 0$ exists, for that*

$$F(v) = \lambda \cdot v \tag{3.3}$$

*Every vector $v \in V$ that is different to the null vector and for that Equation 3.3 holds is called **eigenvector** of F (to eigenvalue λ). [Fis89]*

As our squared adjacency matrix $A \in \mathbb{R}^{n \times n}$ is an endomorphism of the $\mathbb{R}$-vectorspace $\mathbb{R}^n$, a vector $v \in \mathbb{R}^n$ that is different to the null vector and for that a $\lambda \in \mathbb{R}$ with

$$A \cdot v = \lambda \cdot v \tag{3.4}$$

exists, is called eigenvector of A to the eigenvalue λ. Note that matrices can have zero or several eigenvalues and eigenvectors. To calculate the eigenvector centrality of a developer network, an unique eigenvector of its adjacency matrix is needed. As the adjacency matrix has only nonnegative entries, the Perron–Frobenius theorem states, that at least one nonnegative eigenvector exists if the network has no independent groups.

Theorem 1. *Perron–Frobenius Theorem: If all elements $(A)_{ij}$ of an irreducible matrix A are nonnegative, then $\lambda = minE_\gamma$ is an eigenvalue of A and all the eigenvalues of A lie on the disk $|z| \leq \lambda$, where, if $\gamma = (\gamma_1, \gamma_2, ..., \gamma_n)$ is a set of nonnegative numbers (which are not all zero),*

$$E_\gamma = inf \left\{ \mu : \mu\gamma \geq \sum_{j=1}^{n} |a_{ij}|\gamma, 1 \leq i \leq n \right\} \tag{3.5}$$

[Weib]

Remark 1. *As E_γ only contains elements that are nonnegative, at least one positive eigenvalue of an irreducible, nonnegative matrix A exists.*

Definition 3.2.3. *A square $n \times n$ matrix $A = (A)_{ij}$ is called* ***reducible*** *if the indices $1, 2, \dots, n$ can be divided into two disjoint nonempty sets $i_1, i_2, \dots, i_\mu$ and $j_1, j_2, \dots, j_\nu$ (with $\mu + \nu = n$) such that*

$$a_{i_k j_l} = 0, \quad k = 1, 2 \dots, \mu \text{ and } l = 1, 2, \dots \nu \tag{3.6}$$

A matrix is reducible if and only if it can be placed into block upper-triangular form by simultaneous row/column permutations. In addition, a matrix is reducible if and only if its associated directed graph is not strongly connected.

A square matrix that is not reducible is said to be ***irreducible****. [GR]*

Definition 3.2.4. *A* ***strongly connected*** *graph is a directed graph in which it is possible to reach any node starting from any other node by traversing edges in the direction(s) in which they point. The nodes in a strongly connected digraph therefore must all have indegree of at least 1 [Weic].*

Remark 2. *A matrix with an associated directed graph is irreducible if and only if it is possible to reach any node starting from any other node by traversing edges.*

Remark 3. *Undirected graphs can easily be transformed to directed graphs by replacing every undirected edge with directed edges for each direction. Therefore Remark 2 holds analogously for undirected graphs. Furthermore an undirected graph is irreducible if and only if it is connected.*

As usually OSS projects do not consist of completely independent groups of developers, the graphs of the associated developer networks are (strongly) connected and, at least, one positive eigenvalue to the adjacency matrix exists. For the calculation of eigenvector centrality, the largest positive eigenvector of the adjacency matrix is chosen. As this eigenvector is unique (i.e, it is not possible that two largest positive eigenvalues exist), eingevector centrality can be calculated with it. The chosen eigenvector $v \in \mathbb{R}^n$ has n entries, one entry for every node in the developer network. As the eigenvector contains compressed information about the adjacency matrix, and therefore also about the connections in the developer network, the entries give direct information about the node centrality in the developer network and thereby about the importance of the developers in the OSS project. The higher the absolute value of an entry in the eigenvector is, the more important is the corresponding node in the developer network. Again, developer which nodes have high results in all three developer networks are identified as core developers.

But both of this methods have the same disadvantage in the representation of the OSS project. Even through they are able to identify accurate core developers, there is one thing they do not consider in their calculations: the influence different types of interactions have on each other. In OSS development the different interactions of developers are not independent of each other. For example if a developer opens an issue about a software problem, it will directly influence that another developer tries to fix it. In the developer network approach, the developer that opens the issue just gets a new connection in the issue network, the developer that fixes the issue just gets a connection in the cochange network. A dependence between them is not apparent. Therefore could it be possible, that a centrality measurement which includes this dependence in its calculation, is able to identify core developers even more accurate.

3.3 Centrality Measurement in Multiplex Networks

A way to consider the influence that for example cochange and issue activities have on each other, is to use multiplex networks and multiplex centrality measures for the identification of core developers. But this approach has not been applied on OSS projects yet. In the following section we introduce a multiplex centrality measurement that has been used on similar networks but in a different area.

3.3.1 EDCPTD Centrality

A centrality measurement that is able to consider dependencies between different types of interactions is EDCPTD centrality. EDCPTD centrality stands for **E**ssential nodes **D**etermining based on **CP** **T**ensor **D**ecomposition [DWZ17] and is a quite new centrality measurement, that is currently primarily used for real-world biological networks. This real world-biological networks have, similar to the developer network model of OSS projects, biological objects as nodes and interactions between the different objects as edges between the corresponding nodes. For example, in the yeast landscape multilayer network (YLMN) are proteins represented by nodes and different interactions and correlations of proteins represented by different edges between the corresponding nodes. On the real-world biological networks, EDCPTD centrality is able to consider the influence that different types of interactions have on each other and therefore can calculate the centrality of the nodes more accurate than degree- and eigenvector centrality do [DWZ17].

To include the influence between different interactions in its calculation, EDCPTD centrality is not calculated separately for every type of interaction, such as degree- and eigenvector centrality, but once already including all types of interactions. Therefore it does not use separate networks with corresponding adjacency matrices for every type of interaction, but a multiplex network that includes all types of interactions in one network. Different types of interactions are now represented by connections on different layers in the multiplex network instead of connections in different single-layer networks. Because of this, EDCPTD centrality is called a multiplex centrality measurement.

Definition 3.3.1. *A **multiplex network** is a set of n nodes interacting in m layers, each reflecting a distinct type (or time or resolution) of interaction linking the same pair of nodes [MRP+14].*

Similar to the mathematical representation of a single-layer network with an adjacency matrix, a multiplex network can be represented by a tensor.

Definition 3.3.2. *An k^{th}-order tensor in d-dimensional space is a mathematical object that has k indices and d^k components and obeys certain transformation rules. Each index of a tensor ranges over the number of dimensions of space. However, the dimension of the space is largely irrelevant in most tensor equations (with the notable exception of the contracted Kronecker delta). Tensors are generalizations of scalars (that have no indices), vectors (that have exactly one index), and matrices (that have exactly two indices) to an arbitrary number of indices. [TR]*

In order to apply EDCPTD centrality on them, the real-world biological networks were represented by a fourth-order tensor $M \in \mathbb{R}^{n \times m \times n \times m}$, where n is the number of nodes and m is the number of layers. An entry in this tensor M is defined as follows:

Definition 3.3.3.

$$(M)_{i\alpha j\beta} = \begin{cases} w_{i\alpha j\beta} & \textit{if node } i \textit{ on layer } \alpha \textit{ is connected to node} \\ & j \textit{ on layer } \beta \textit{ with weight } w_{i\alpha j\beta} \\ 0 & \quad \textit{otherwise} \end{cases} \tag{3.7}$$

[DWZ17]

This forth-order tensor does not only has the advantage that the influence between connections on different layers can be retraced, but it also has space for entries that represent a direct connection between two layers. This could for example make sense if a node on one layer has direct influence on a node on another layer. To distinguish between this type of connections and the connections on one layer, we introduce the terms intra-layer connection and inter-layer connection. This differentiation further helps to evaluate, if the use of inter-layer connections is reasonable for centrality measurements.

Remark 4. *A connection between developers on the same layer is called **intra-layer connection**. Such a connection is represented by a tensor entry $(M)_{i\alpha j\beta}$ with $\alpha = \beta$.*

Remark 5. *A connection between developers on different layers is called **inter-layer connection**. Such a connection is represented by a tensor entry $(M)_{i\alpha j\beta}$ with $\alpha \neq \beta$.*

EDCPTD centrality is based on the idea to get compressed information about the connections in the tensor by decomposing it. Therefore, as the name already suggests, CP tensor decomposition is used. CP tensor decomposition is the CANDECOMP/PARAFAC or just canonical polyadic kind of tensor decomposition. It approximates a tensor as a finite sum of vectors concatenated with the vector outer product.

Definition 3.3.4. *CP tensor decomposition is an approximation that factorizes a tensor into a finite sum of outer products of vectors. For a k^{th}-order-tensor M an best approximation $\hat{M}$ of the form*

$$\hat{M} = \sum_{r=1}^{R} \lambda_r a_r^{(1)} \circ a_r^{(2)} \circ \cdots \circ a_r^{(n)} \approx M \tag{3.8}$$

with approximation rank $R > 0$ and singular values $\lambda_1, ..., \lambda_R \in \mathbb{R}$ of M, $\lambda_1 \geq ... \geq \lambda_R$ is calculated. [Gos]

Remark 6. *Let V be a normed space of real-valued functions that are defined over $[a, b]$, U a set of real-valued functions that are defined over $[a, b]$ and $\| \cdot \|$ a norm over $V \cap U$. An element $f \in V$ is called **best approximated** by $u \in U$ if*

$$\|f - u\| \leq \|f - v\| \quad \forall v \in U \tag{3.9}$$

holds. [Joh]

Remark 7. *The entry $(\hat{M})_{i_1,i_2,...,i_n}$ of the CP decomposition of a n-order-tensor can be written as*

$$(\hat{M})_{i_1,i_2,...,i_n} = \sum_{r=1}^{R} \lambda_r (a_r^{(1)})_{i_1} \cdot (a_r^{(2)})_{i_2} \cdot \cdots \cdot (a_r^{(n)})_{i_n} \approx (M)_{i_1,i_2,...,i_n} \tag{3.10}$$

[Gos]

For the four dimensional tensor $M \in \mathbb{R}^{n \times m \times n \times m}$ the result of the CP tensor decomposition $\hat{M}$ has the form:

$$\hat{M} = \sum_{r=1}^{R} \lambda_r a_r^{(1)} \circ a_r^{(2)} \circ a_r^{(3)} \circ a_r^{(4)} \approx M \tag{3.11}$$

[DWZ17]

With the desired approximation rank R, the singular vectors $\lambda_1, ..., \lambda_R \in \mathbb{R}$ of M and the to the singular values corresponding singular vectors $a_r^{(1)}, a_r^{(3)} \in \mathbb{R}^n$ and $a_r^{(2)}, a_r^{(4)} \in \mathbb{R}^m$. For the singular values holds $\lambda_1 \geq ... \geq \lambda_R$.

The desired approximation Rank R determines how exact the approximation of the tensor shall be calculated. The singular values and singular vectors of tensors are the interesting result of the decomposition, as they contain compromised information about the structure of the tensor. This information can be used to determine the centrality of nodes. This concept reminds of the way eigenvector centrality works. More in detail are eigenvalues a special form of singular values and eigenvectors a special form of singular vectors. So the basis for EDCPTD centrality is quite similar to the basis for eigenvector centrality.

As already in eigenvector centrality, only the singular vectors corresponding to the largest singular value λ_1 are considered for the centrality calculation. But different to the eigenvector centrality, four singular vectors correspond to the largest singular value. $a_1^{(1)}$ and $a_1^{(3)}$ with n entries and $a_1^{(2)}$ and $a_1^{(4)}$ with m entries. These singular vectors give direct information about the centrality of the nodes and layers [DWZ17]. The vectors $a_1^{(1)}$ and $a_1^{(3)}$ provide the so called hub and authority scores of the nodes. Analogous do the vectors $a_1^{(2)}$ and $a_1^{(4)}$ provide the hub and authority scores of the layers. The hub scores reveal how often connections on a node or a layer take place. The authority scores reveal how important these connections are. With this scores it is possible to calculate the EDCPTD centrality for every node.

Definition 3.3.5. *For a forth-order tensor $M \in \mathbb{R}^{n\times m\times n\times m}$ with a CP decomposition $\hat{M} = \sum_{r=1}^{R} \lambda_r a_r^{(1)} \circ a_r^{(2)} \circ a_r^{(3)} \circ a_r^{(4)}$ where $a_r^{(1)}, a_r^{(3)} \in \mathbb{R}^n$ and $a_r^{(2)}, a_r^{(4)} \in \mathbb{R}^m$ for $r = 1, 2, \dots R$ the EDCPTD score for every node i can be calculated as follows:*

$$H_i = \frac{1}{2}\sum_{l=1}^{m} |(a_1^{(1)})_i (a_1^{(2)})_l| + |(a_1^{(3)})_i (a_1^{(4)})_l| \tag{3.12}$$

[DWZ17]

This returns a score between zero and one for every node. If the scores from all nodes are added together, the total value will be one. The higher the score of one node is, the more important is the node. As all four singular vectors give different information about the centrality of nodes and layers, they are all necessary for the calculation of EDCPTD centrality. Therefore the multiplex network has to be modeled as forth-order tensor, independently of the plausibility of inter-layer connections in the application.

This multiplex centrality measurement is able to identify central nodes, such as proteins, cells or genes, in real-world biological networks. As evaluated by Wang et al. ([DWZ17]) is the result of EDCPTD centrality in this networks more accurate than the results of degree- and eigenvector centrality. In this thesis we will evaluate if this multiplex centrality measurement is also applicable to OSS projects and if it is able to identify core developers accurate.

4 Research Questions

The centrality measures degree- and eigenvector centrality, that are used to identify core developers in OSS projects, have the disadvantage of not considering dependencies between different types of interactions. Such correlations do exist in OSS projects, as for example issue and mail activities do in many cases also cause changes on the code. In this point, the accuracy of the identification of core developers can still be improved.

Therefore we examine the new approach of applying the multiplex centrality measurement EDCPTD centrality on OSS projects. When EDCPTD centrality is applied on real-world biological networks, it is able to consider dependencies between different layers in the network and determine essential nodes more accurate than degree- and eigenvector centrality can. For the application on OSS projects and the identification of core developers we expect a similar result. With the use of a multiplex centrality measurement, the influence that code contributions and mail and issue activities have on each other shall be considered for the identification of core developers. The modeling of a OSS project as a multiplex network causes, that indirect influence between connections on different layers can be retraced. Further with the introduction of inter-layer connections, even direct influence between layers can be added to the model. For example if a developer reviews someones pull-request, a direct connection between the reviewer on the issue layer and the committer on the cochange layer could be added to the multiplex network to represent the influence the review has on the code. In this way dependencies in OSS projects can be considered more accurate to identify core developers.

To evaluate this expectation, we will model OSS projects as multiplex networks and apply EDCPTD centrality on them. To ensure, that the developers with the highest EDCPTD score a correctly identified core developer, we will examine two research questions.

(RQ1) Is EDCPTD centrality able to identify core developers that have also been rated as important by established centrality measures?

To evaluate this question, we will apply EDCPTD centrality as well as degree- and eigenvector centrality on real OSS projects and compare the results. To figure out if EDCPTD centrality is able to identify core developers, a correlation between a high EDCPTD score of a node and an important role of the corresponding developer in the project has to be shown. As degree- and eigenvector centrality are established methods to identify the most important developers in cochange, mail and issue interactions, their results can be used to validate the results of EDCPTD centrality. It can be assumed, that a core developer of the OSS project is either one of the most important developers for code contributions, on the mailing list or in issue activities or is surpassing important in all three of them. Therefore we will apply degree- and eigenvector centrality on the three one-dimensional developer networks, that represent the common code contributions, the communication on the mailing list and activities on issues. Furhter we will union these three developer networks, to one network that represents all interactions in the same way, and apply the two centrality measures on it. If a developer that has been identified as important by EDCPTD centrality has also been identified as important by one of the other eight measures, it can be concluded that the developer was identified by EDCPTD centrality correctly.

(RQ2) Is EDCPTD centrality able to identify core developers that fulfill the typical behavior of core developers?

To evaluate this question, we will test if EDCPTD centrality is able to detect the patterns of behavior from core developers that where described inSection 2.2 and rates the developers that fulfill them as most important. Therefore we will generate synthetic OSS projects in which theses patterns occur. The developers that fulfill these patterns are labeled as core developers. When EDCPTD centrality is applied on these artificial networks, the labeled core developer shall be the developers with the highest EDCPTD score. If this is the case, it will indicate that EDCPTD centrality is able to detect and rate these patterns correctly.

If both both research questions confirm the expected result, it can be assumed that EDCPTD centrality is able to identify core developers in OSS projects.

5 Methodology

In this chapter we will explain how OSS projects can be transformed to multiplex networks and how EDCPTD centrality can be calculated on them. Further we state how the transformation and the calculation of EDCPTD centrality can be implemented in R. In the last two sections the organization and execution of the two research questions is described.

5.1 OSS Projects Modeled as Multiplex Networks

Before the two research questions from the previous chapter can be examined, OSS projects have to be represented in an appropriate model such that EDCPTD centrality can be applied on them. Therefore the data from OSS projects has to be transformed to a multiplex network. Similar to the single-layer network model the developers that are active in the OSS project can be represented as nodes in the multiplex network. Also quite similar to the single-layer networks, the different interactions between developers can be represented by edges between the corresponding nodes. But this time, different interactions are not modeled in different single-layer networks but on different layers in the multiplex network (Figure 5.1).

5.1.1 Forth-Order Tensor Representation

Further this multiplex network has to be represented as forth-order tensor $M \in \mathbb{R}^{n \times m \times n \times m}$, where n is the number of developers and m is the number of different interactions. Similar to the single-layer networks that where described in Section 3.1 we consider three different layers to represent the OSS project: layer 1 to represent the cochange interactions, layer 2 to represent the email interactions and layer 3 to represent the issue interaction. So in this application $m = 3$ holds. An entry in this forth-order tensor representation of an OSS project can be defined as follows:

$$(M)_{i\alpha j\beta} = \begin{cases} w_{i\alpha j\beta} & \text{if developer } i \text{ on layer } \alpha \text{ has interacted} \\ & \text{with developer } j \text{ on layer } \beta \ w_{i\alpha j\beta} \text{ times} \end{cases} \tag{5.1}$$

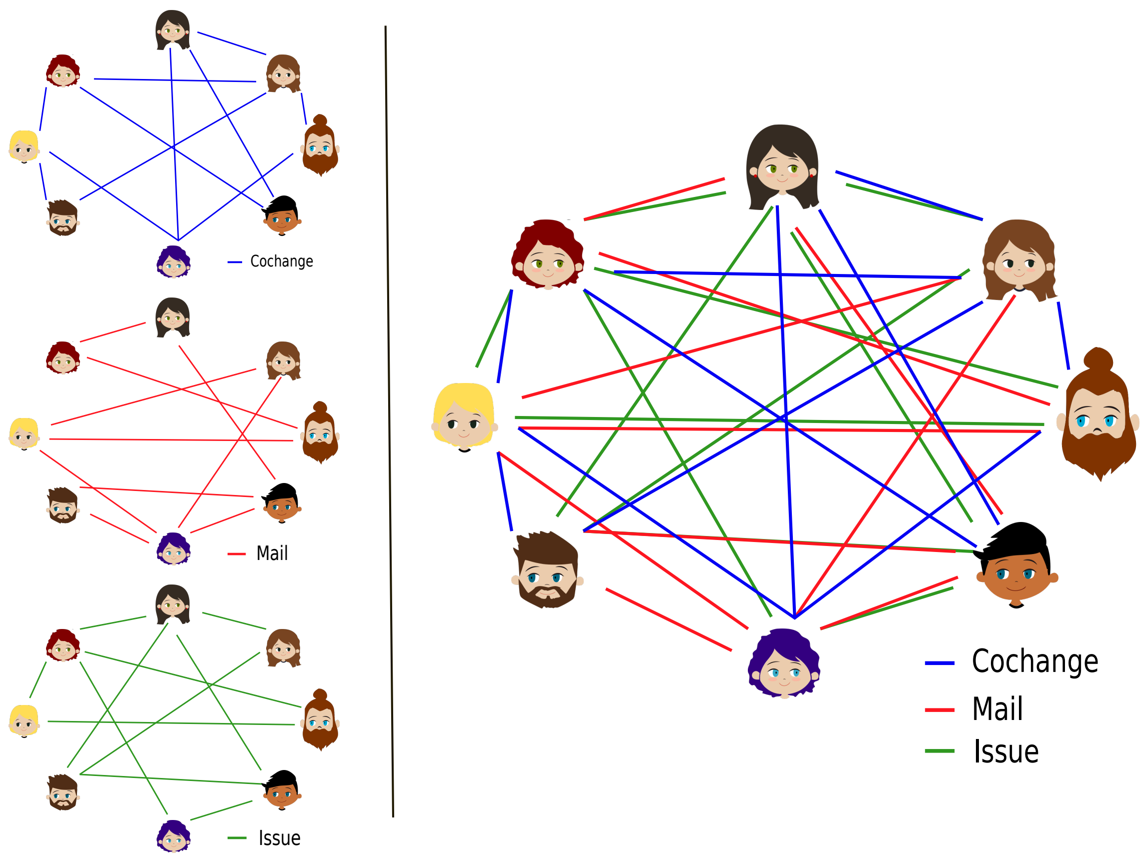

Figure 5.1: Example developer networks. On the left side are three single-layer networks for the cochange, mail and issue activities. On the right side is one multiplex network that represents the same connections. This multiplex network has only intra-layer connections.

With this forth-order tensor, all information that previously has been modeled in independent single-layer networks, are now represented in one forth-order tensor. All information are now compact represented in one multiplex network. This has the consequence, that that indirect influence between two types of interaction can be retraced. For example if developer A communicates with developer B on the issue layer and developer B works on the same code as developer C, an indirect influence of developer A on developer C can be considered in the centrality measurement. This indirect influence could be interesting to identify core developers. For example if developer A suggests a new functionality in an issue and assigns developer B to it, the implementation of this new functionality from developer B could have the consequence that developer C updates other methods in the same code file to fit this new functionality. In this way the suggestion of developer A has influenced the work of developer C, even though they have not been active in the same type of interaction.

Even further, the representation of the multiplex network as forth-order tensor offers the possibility to represent direct influence between developers on different layers. With the introduction of inter-layer connections, relations between different types of interactions can be modeled. This can for example be useful in the representation of OSS projects, if one developer reviews another ones code on GitHub. Now these developers can not only be linked with an intra-layer connection on the issue layer, but also with an inter-layer connection between the reviewer on the issue layer and the committer on the cochange layer. So the direct influence between the issue layer and the cochange layer can become part of the multiplex network. Another application of inter-layer connections in OSS projects would be if the text of a mail contains the hash code of a commit. A connection that represents these interaction between the developer that wrote the mail on the mail layer and the developer that contributed the commit on the cochange layer can be added to the multiplex network.

In this way the transformation of an OSS project in a multiplex network with forth-order tensor representation has the advantage that direct and indirect connections between different types of developer interactions can be considered for centrality measurements. So the multiplex network model enables a more accurate representation of an OSS project than it was possible with the single-layer network approach. To evaluate if the results of the multiplex centrality measurement EDCPTD are also more accurate than the results of degree- and eigenvector centrality, this theoretically transformation to a multiplex network and the calculation of EDCPTD centrality has to be implemented.

5.1.2 Normalized Forth-Order Tensor Representation

But before our implementation is explained, we will introduce another approach to represent the OSS project as forth-order tensor, the normalized forth-order tensor representation.

In our first EDCPTD centrality calculations, we analyzed an OSS project which had much more interactions for the cochange layer than for the other two layers. The developers with the highest EDCPTD centrality have been exactly the same developers that where the most important developers in the cochange single-layer network. The developers that where important in the mail and issue single-layer networks have only rarely been identified as important by EDCPTD centrality. So the cochange layer dominated the calculation of EDCPTD centrality. But, for example a high number of code contributions does not implicate that the mails from the mailing lists and the issue activities are less important. Even in opposite the communication could be more important to organize the huge number of code contributions. So the domination of the centrality measurement by the most active layer could be an unwanted effect.

Therefore we do not only consider the ordinary forth-order representation M of the multiplex network, that was introduced in Section 5.1.1, in our analysis, but additionally we introduce a normalized forth-order tensor representation M'. In this representation the weights of the connections are adjusted so that the sum of the weight of all connections on every layer, and between every two layers are the same. More precisely if all weights of connections on a single layer or all weights of connections between two layers are summed up together the result will always be one.

This is achieved by dividing the weight of a connection by the sum of the weights of all connections on this layer. The same happens for inter-layer connections. The weight of one inter-layer connection is divided by the sum of all inter-layer connections between these two layers. In this way the connections and the proportion of the weights of the connections between two nodes on a layer (or inter-layer) stays the same but the weightiness of the layers, and thereby also their importance in the centrality measurement, is equalized. An entry in this normalized fourth-order tensor M' can be calculated by using the ordinary forth-order tensor M as follows:

$$(M')_{i\alpha j\beta} = \frac{(M)_{i\alpha j\beta}}{\sum_{k=1}^{n}\sum_{l=1}^{n}(M)_{k\alpha l\beta}} \tag{5.2}$$

EDCPTD centrality can be calculated on this normalized forth-order tensor in the same way as on the ordinary forth-order tensor. All analyses in this bachelor thesis will be executed twice, as well with the ordinary forth-order tensor representation as with the normalized forth-order tensor representation. In this way we will evaluate if the normalized forth-order tensor is a more reasonable representation for centrality measurement than the ordinary forth-order tensor.

5.2 Implementation of EDCPTD Centrality

In theory it is possible to apply EDCPTD centrality on OSS projects represented by a (ordinary or normalized) forth-order tensor. To evaluate the research questions and to show that EDCPTD centrality is a multiplex centrality measurement that identifies core developer in OSS projects, the forth-order tensor representation, the CP decomposition and the EDCPTD centrality calculation have to be implemented.

For the implementation of the forth-order tensor and its CP-decomposition we make use of the statistical software environment R[1] and the package rTensor[2]. R is a programming language and environment that is made for statistical computing and to illustrate graphics. It is a suitable software environment to implement the forth-order tensor modeling and the EDCPTD centrality as it is can handle and store data effectively and provides many operators for calculations on arrays and matrices[3].

With R it is easy to create a forth-order tensor representation of a given multiplex network from an OSS project. The following code creates an array `M.arr` $\in \mathbb{R}^{n\times m\times n\times m}$ with only zeros in it. So an array that already has the needed dimensions but does not represent any connections is saved.

```
M.arr = array(0, dim = c(n, m, n, m))
```

In the next step, all connections from the multiplex network of the OSS project that shall be represented by the forth-order tensor have to be entered to this array. Therefore analogous to Equation 5.1 the entries can be assigned:

```
M.arr[i, alpha, j, beta] = w.i.alpha.j.beta
```

[1]https://www.r-project.org/ (Accessed at 19.09.2019)
[2]https://cran.r-project.org/web/packages/rTensor/index.html (Accessed at 19.09.2019)
[3]https://www.r-project.org/about.html(Accessed at 19.09.2019)

In this R code snippet is the weight `w.i.alpha.j.beta`, that represents the connection between developer i on layer α to developer j on layer β in the multiplex network, assigned to the array entry $(\texttt{M.arr})_{i\alpha j\beta}$. These weights of the connections between developers have to be extracted from OSS projects. When all weights are known, an array that has the same dimensions and equal entries as the forth-order tensor representation of the multiplex network can be created in R.

For the normalized forth-order tensor representation it is the easiest way to normalize this array. Analogous to Equation 5.2 the entries can be assigned in R with:

```
norm.M.arr[i, alpha, j, beta] =(M.arr[i, alpha, j, beta] / sum ( M.arr[, alpha, ,
    beta]))
```

Now also an array with the same dimensions and equal entries as the normalized forth-order ternsor representation can be created.

But to calculate EDCPTD centrality, an array and the normal R operators are not sufficient, as they cannot decompose tensors. Therefore the R package rTensor is needed. rTensor provides a set of tools for the creation, the manipulation, and the modeling of tensors with arbitrary number of modes[4]. Therefore it provides a class 'Tensor' that wraps around an array. On objects of this class tensor operations can be applied. In order to apply these operations on the previous created arrays, they have to be transformed to objects of the 'Tensor' class. This can be achieved in R as follows:

```
M <- rTensor::as.tensor(M.arr)
norm.M <- rTensor::as.tensor(norm.M.arr)
```

Now forth-order tensors $\texttt{M} \in \mathbb{R}^{n\times m\times n\times m}$ and $\texttt{norm.M} \in \mathbb{R}^{n\times m\times n\times m}$ that represent the multiplex networks of the OSS project have been created in R .

To calculate EDCPTD centrality, CP decomposition has to be applied on this tensors. The package rTensor provides a method that calculates such a decomposition of tensor objects. The following R code snippet calculates a CP decomposition as defined in Equation 3.11:

```
M.hat = rTensor::cp(M, num_components = 1, max_iter = 50, tol = 1e-05)
```

In this method the parameter `num_components` indicates how many singular vectors shall approximate one order of the tensor. To calculate EDCPTD centrality, `num_components` will be chosen to be 1, as only the singular vectors corresponding to the largest singular value are needed. The parameter `max_iter` and `tol` decide how exact the approximation of the tensor shall be. Here `max_iter` indicates how many iterations to calculate the CP decomposition shall be made as maximum and the parameter `tol` indicates the maximum relative error in the Frobenius norm the approximation of the tensor shall have. The CP decomposition method stops its iterations, if the relative error is below `tol` or the maximum of `max_iter` iterations is reached. For our calculations we use the default values `max_iter = 50` and `tol = 1e-05`. Analogous a decomposition for `norm.M` can be created.

[4]https://cran.r-project.org/web/packages/rTensor/index.html(Accessed at 19.09.2019)

The calculated `M.hat` has all information needed to calculate EDCPTD centrality in R. The four needed singular vectors are saved by `M.hat` in a list `"U"`. A possible function that uses `M.hat` to calculate the EDCPTD scores for all developers defined in Equation 3.12 and returns them in an array could look like this:

```r

calculate.EDCPTD.scores = function(M.hat){

EDCPTD.scores  = array(0, dim = n)

for(user in 1:n){

        # sum over the layers
        for(layer in 1:m) {

                EDCPTD.scores[user] = (EDCPTD.scores[user]
                  + abs(M.hat[["U"]][[1]][user]*M.hat[["U"]][[2]][layer])
                  + abs(M.hat[["U"]][[3]][user]*M.hat[["U"]][[4]][layer]))

    }

}

# the division in front of the sum
result = EDCPTD.scores / 2

return(result)
}
```

Now it is possible to model the OSS project as a multiplex network with forth-order tensor representation and calculate EDCPTD centrality in `R`. In the next step the research questions can be evaluated.

5.3 (RQ1) EDCPTD Centrality Compared to Already Established Centrality Measures with the Help of Real OSS Projects

The first research question *(RQ1)* shall figure out if the core developers, which the multiplex centrality measurement EDCPTD centrality identifies, are really important developers in OSS projects. To examine this we will apply EDCPTD -, degree - and eigenvector centrality on real OSS projects and compare the results. As degree- and eigenvector centrality are already established centrality measures for OSS projects, a comparison of their result with the results of EDCPTD centrality will give information about the reliability of the results of EDCPTD centrality. To conduct this comparison, data from real OSS projects are needed.

5.3.1 Retrieval of OSS Project Data

For the comparison of the centrality measurements we extract data from several OSS projects. In particular for every OSS project that shall be analyzed the commit history, the mails from the mailing lists and the issue activities are necessary to model the multiplex network.

Because the source codes of OSS projects are public, it is easy to get the commit histories via the public GIT[5] repositories of the projects. Access to the mailing lists is possible with the public archive GMANE[6]. GMANE is used as a gateway that enables to read mails without being part of the mailing lists. The issue data can be retrieved with the help of the API of the particular platform, such as the GITHUB REST API[7] or the JIRA REST API[8] . So all needed data can be accessed. But the data still needs to be processed to a practical and consistent format.

For processing of the data, we use the frame-work CODEFACE[9] developed by Siemens and its extension CODEFACE-EXTRACTION[10]. These tools extract the needed meta data, such as the sender and the receiver of an email, from the originally retrieved raw data. Further it makes the meta data from the different sources consistent, for example if one user from the commit history and one user from the mailing list have the same username and the same email address they are combined to one user. The processed meta data now can finally be used by another tool, CORONET[11], that is able to build socio-technical developer networks of OSS projects. CORONET can for example build developer networks as described in Chapter 3. For this network representation it uses, consistent to our implementation in Section 5.2, the statistical programming language R and the R package IGRAPH[12].

5.3.2 Data for The Comparison of OSS Projects

To examine the first research question, we use the data from three different OSS projects. These are APACHE ZEPPELIN, OPENSSL and OWNCLOUD. All three provide data about the cochange, the mail and the issue interactions. The data we extracted covers several years of software development. During this time the projects grew and advanced. So do all three projects have during this time an increasing number of developers. Thereby also the number of interactions between developers increases. Interactions that took place in different stages of the project, might have a different importance. In order to analyze projects with a consistent structure, we split the project data in 6-month time ranges.

The OSS project APACHE ZEPPELIN[13] is a web-based notebook that enables interactive data analytics. For our research we analyze the developer activities from the time between 20.03.2015 and 19.03.2017. The data about the project in this time is divided in four time ranges, each covering 6 months. The number of active developers in the project increased in this time. In the first range 55 developers participated in the project. In the last time range 169 developer have been active.

The second OSS project, OPENSSL[14], is a toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. For the analysis the developer

[5]https://git-scm.com/(Accessed at 20.09.2019)
[6]http://home.gmane.org/(Accessed at 20.09.2019)
[7]https://developer.github.com/v3/(Accessed at 20.09.2019)
[8]https://developer.atlassian.com/server/jira/platform/rest-apis/(Accessed at 20.09.2019)
[9]https://github.com/siemens/codeface(Accessed at 20.09.2019)
[10]https://github.com/se-passau/codeface-extraction(Accessed at 20.09.2019)
[11]https://github.com/se-passau/coronet(Accessed at 20.09.2019)
[12]https://igraph.org/r/(Accessed at 20.09.2019)
[13]https://zeppelin.apache.org/(Accessed at 20.09.2019)
[14]https://www.openssl.org/(Accessed at 20.09.2019)

activities between 14.03.2002 and 14.03.2019 are examined. This data is split into 34 time ranges. Again the number of active developers in this project increases by time. In the first range only 11 developers have been active. In the last range 322 active developers contributed to the project.

The third and last project we will analyze is the file hosting service OWNCLOUD[15]. For this analysis we consider the developer activities between 25.08.2012 and 09.03.2019. The data is split into 13 time ranges, again each with the length of 6 month. Also in this OSS project the number of active developers increases by time. In the first time range 61 developers have been active. In the last time range this number raised to 258 developers.

5.3.3 Calculation of the Established Centrality Measures

The data of these projects is processed as described in Section 5.3.1 and in this way imported to CORONET. With the help of CORONET this data can be split into the 6-month time ranges and developer networks for every time range can be constructed. For every time range one single-layer network for the cochange activities, one single-layer network for the mail activities and one single-layer network for the issue activities is constructed. Additionally, we analyze one single-layer union network, obtained by uniting the cochange, mail and issue network. This shall give a comparison if it has a reasonable advantage to model the independent single-layer networks to a more complex multiplex network instead of uniting them to an overall single-layer network. All networks that we build are weighted and undirected.

For all four of these networks, degree- and eigenvector centrality can be calculated. As CORONET generates the single-layer networks as IGRAPH objects, the degree- and eigenvector centrality methods of this R package can easily be applied on them. For both centrality measures IGRAPH provides a method that calculates the centrality score for every node of an IGRAPH object:

```
degree.centralities = igraph::degree(developer.network)
eigenvector.centrality = igraph::eigen_centrality(author.networks[[x]], directed =
    TRUE)
```

In this way, centrality scores of the already established centrality measures can be obtained from OSS projects. This centrality scores shall be compared to the centrality scores determined by EDCPTD centrality.

[15]https://owncloud.org/(Accessed at 20.09.2019)

5.3.4 Calculation of EDCPTD Centrality

For the comparison of the different centrality measures, the EDCPTD centrality scores for the developers are also needed. Therefore the data of the three OSS projects have to be transformed to forth-order tensors. With the help of CORONET the adjacency matrices of the previous in Section 5.3.3 created single-layer networks can be obtained. Given the adjacency matrices A_1 of the cochange network, A_2 of the mail network and A_3 of the issue network, it is easy to transfer the entries in the a forth-order tensor $M \in \mathbb{R}^{n \times m \times n \times m}$ as follows:

$$(M)_{i1j1} = (A_1)_{ij} \tag{5.3}$$

$$(M)_{i2j2} = (A_2)_{ij} \tag{5.4}$$

$$(M)_{i3j3} = (A_3)_{ij} \tag{5.5}$$

Unfortunately it is not possible yet, to obtain information about inter-layer connections with the retrieval of data as in Section 5.3.1. Therefore the modeling of the OSS project is limited to multiplex networks with only intra-layer connections. So the forth-order tensor that is built shall have exactly the same entries as the three adjacency matrices of the cochange, the mail and the issue singe-layer networks. In the second research question we also examine the influence that the usage of inter-layer connection has on the centrality measurement.

With `coronet` the adjacency matrices can not only be obtained in the common matrix form but also as sparse matrix[16]. This is a memory efficient representation that only saves the entries in the matrix that are unequal zero. This corresponds exactly to the entries that have to be transferred to the forth-order tensor. By using the sparse matrices, run-time expensive iterations about the whole matrices are not necessary. But the structure of these sparse matrices is a bit more complicated than the structure of common array-like matrices. Instead of a two-dimensional array, a sparse matrix saves three lists of the same size. A list '`x`' with the weights of the entries in the matrix that are unequal zero, a list '`i`' with the first indices of that entries and a list '`j`' with the second indices of that entries. For example the first entry in list `x` represents the weight of the entry in the matrix with the coordinates of the first entries in list `i` and list `j`. Additionally to this structure the indices that are saved in the lists `i` and `j` start at zero, while the indices of an array start at one. Therefore the indices in the lists `i` and `j` of the sparse matrix have to be incremented by one before the entries can be transferred to the array. If this is taken in account the three given sparse adjacency matrices `adjacency.cochange`, `adjacency.mail` and `adjacency.issue` can be transferred into a four-dimensional array with following function:

[16]https://www.rdocumentation.org/packages/Matrix/versions/1.2-17/topics/sparseMatrix(Accessed at 20.09.2019)

```r
create.multiplex.array = function(adjacency.cochange, adjacency.mail, adjacency.
    issue){

  # create array
  M.arr = array(0, dim = c(n, m, n, m))

  # transfer the entries from the cochange adjacency matrix
  for(entry in 1:length(adjacency.cochange)){
    i = adjacency.cochange@i[entry] + 1
    j = adjacency.cochange@j[entry] + 1
    M.arr[i, 1, j, 1] = adjacency.cochange@x[entry]
  }

  # transfer the entries from the mail adjacency matrix
  for(entry in 1:length(adjacency.mail)){
    i = adjacency.mail@i[entry] + 1
    j = adjacency.mail@j[entry] + 1
    M.arr[i, 2, j, 2] = adjacency.mail@x[entry]
  }

  # transfer the entries from the issue adjacency matrix
  for(entry in length(adjacency.issue)){
    i = adjacency.issue@i[entry] + 1
    j = adjacency.issue@j[entry] + 1
    M.arr[i, 3, j, 3] = adjacency.issue@x[entry]
  }

  return(M.arr)
}
```

Now a four-dimensional array that has equal dimensions and the same entries as the ordinary forth-order tensor representation of the multiplex network has been created. With the procedure of Section 5.2 also a normalized array can be created, the two arrays can be transformed to an object of the S4 tensor class and EDCPTD centrality can be calculated on both of them.

Now all values that are necessary for the comparison of EDCPTD centrality with already established centrality measurements are given. So the first research question can be evaluated.

5.3.5 Comparison of EDCPTD Centrality with Established Centrality Measures

With the procedures from Section 5.3.3 and Section 5.3.4 the centrality scores of degree- , eigenvector- and EDCPTD can be calculated for every developer in the OSS projects. For degree- and eigenvector centrality, separate scores of the cochange, mail and issue single-layer networks and for the union of these single-layer networks are returned. In contrast to that only one centrality score for every developer is given by EDCPTD centrality. In order to see if the EDCPTD centrality is able to identify core developers in OSS projects correctly, the relationship between a high EDCPTD centrality score of a node and an important role in the OSS project of the corresponding developer has to be shown.

Therefore we compare the result of the EDCPTD centrality measurements with the results of the, already for the identification of core developers established, centrality measures degree- and eigenvector centrality. For this comparison we do not consider the exact order of the results but the set of the most important 20% of developers detected by the different centrality measures. In detail we will evaluate how many developers that are rated to be part of the most important 20% with EDCPTD centrality are also rated to be part of the most important 20% by the other measurements. This shall show if the EDCPTD centrality scores are meaningful to evaluate the importance of developers in OSS projects and thereby can be used to identify core developers.

On the one hand the most important 20% nodes of the EDCPTD centrality measurement can be directly compared to the most important 20% nodes of the other centrality measurements. This will return eight results, one result for the EDCPTD centrality compared with degree- and eigenvector centrality, applied on each of the four single-layer networks, the cochange, the mail and the issue network as well with their union. These results give information how similar the results from EDCPTD centrality are to the results from the already established single-layer centrality measures. If the results from EDCPTD centrality have a constantly high accordance to the results of degree- and eigenvector- centrality applied on one of the three single-layer networks cochange, mail or issue, but not to the two others, it indicates that this layer dominates the centrality calculation of EDCPTD centrality. For example if EDCPTD centrality always determines the same top 20% of developers as degree- and eigenvector centrality applied on the mail network do, but only a few developers that are among the top 20% of the cochange and issue networks, it will signify that EDCPTD centrality rates the connections on the mail network as much more important than the connections on the other layers.

Because a core developer usually is important in all types of interaction, all layers shall be weighted with approximately the same importance for the identification of core developers in OSS projects. Further if the results of EDCPTD centrality have a constantly high accordance to the results of degree- and eigenvector centrality of the union network but not to the other single-layer networks, it will indicate that the more complex modeling in the multiplex network considers no extra information compared to the simple union of single-layer networks. Therefore we do not only expect a high accordance in a few of the eight comparisons, but more a high but steady magnitude in all of the results. This will indicate that all types of interaction are considered by EDCPTD centrality.

On the other hand the most important 20% nodes of the EDCPTD centrality measurement are compared to the union of the most important 20% nodes of the other eight centrality measures. More certainly it is compared if a node that is part of the top 20% in the EDCPTD centrality measurement is also part of the top 20% in any of the other measurements. This comparison shall examine that developers that are considered as important by EDCPTD centrality are rightly considered as important. If a developer is a core developer in the OSS project he has to stand out in one of the ways of interaction, and therefore be detected in the centrality measurement on the corresponding single-layer network, or he has to be surpassing active in all ways of interactions, and therefore be detected by the centrality measurements on the union of the single-layer networks. Therefore we expect a high accordance in this overall comparison.

These comparisons are executed on all time ranges of the three OSS projects Apache Zeppelin, OpenSSL and OwnCloud. So in total 51 6-month time ranges of OSS projects are considered. Further these comparisons are executed twice, once for EDCPTD centrality applied on the ordinary forth-order tensor and once for EDCPTD centrality applied on the normalized forth-order tensor.

This comparison of the results from different centrality measures applied on real OSS projects helps to examine the first research question. For the second research question we try a different approach. This time synthetically generated OSS projects are analyzed.

5.4 (RQ2) Detection of Typical Core Behavior by EDCPTD Centrality applied on synthetically generated OSS Projects

A centrality measurement on an OSS project is not automatically able to identify suitable core developers. Core developers are not just the most important developers in the project, they also have a special behavior as described in Section 2.2. To evaluate our second research question *(RQ2)* and test if EDCPTD centrality is able to detect these behaviors and rates them as important, we will construct several synthetic OSS projects that simulate typical core-peripheral interactions. In several settings, we will construct different types of core behavior in OSS projects.

5.4.1 Random Generated Artificial OSS Projects

Each setting consists of an OSS project with 25 developers and three different ways of interactions, the cochange, the mail and the issue activities. Every OSS project is modeled as a multiplex network with 25 nodes and three layers. From the 25 developers, five are labeled as core developers and twenty are labeled as peripheral developers. Every developer is active in every kind of interaction. The multiplex networks are represented by a forth-order tensors, such that EDCPTD centrality can be applied on them. Again, the forth-order tensors will also be normalized, even if for all layers the same estimated number of connections is created. In this way the influence that the normalization has on the result, if all layers already have the same importance can be examined.

To distinguish the constructed settings, we define in which frequency and with which group of developers a developer is allowed to interact. For example, in one setting peripheral developers are only allowed to interact with core developers, while core developers are allowed to interact with everyone. With which exact developer of this group the developer is connected is decided randomly. Also the exact number of connections a developer has is decided randomly. We only define the estimated number of connections that a developer shall have. The exact number is then generated with a normal distribution around this estimated number with a variance of three. So the number of connections a developer has can be this estimated number minus or plus three. The further away the number of connections is from the estimated number, the less probable is its occurrence in the constructed project.

The random generation of the OSS projects has the advantage that all possible cases can occur. To receive a representative result, each synthetic OSS project is generated 10.000 times. For each project it is measured, how many of the five labeled core developers are among the most important five developers identified by EDCPTD centrality. In this way a representative percentage how accurate EDCPTD centrality identifies core developers correctly can be calculated.

Each setting is executed for several cases with different estimated numbers of connections. Further each case is executed three times, once with only intra-layer connections, once with inter-layer connections only for the core developer between every layers and once with inter-layer connections between every layer for all developers. Thereby the difference that the introduction of inter-layer connections makes can be measured. This is especially interesting, as inter-layer connections have not been used to model OSS projects before.

The generation of these synthetic OSS projects is implemented in R. The OSS projects are directly modeled in forth-order tensors as defined in Equation 5.1. In these tensors, the nodes 1-5 are labeled as core developers, and the nodes 6-25 are labeled as peripheral developers. To calculate the amount of connections a developer shall have, a method that returns an integer that is normal distributed around the `estimated.value` with a variance of three is needed. Further as no negative connections are allowed in the forth-order-tensor this method shall guarantee that no results beyond the variance are returned. The R method we use is:

```r
n.random = function(estimated.value){

  standard.deviation = sqrt(3)

  result = round(rnorm(1, estimated.value, standard.deviation))

  if(result < estimated.value-3){
    result = floor(estimated.value-3)
  } else if (result > estimated.value + 3){
    result = ceiling(estimated.value + 3)
  }

  return(result)
}
```

To avoid a negative number of connections in the constructed settings, the `estimated.value` has to be at least three.

Now random generated synthetic OSS projects can be constructed. The first projects that we construct are projects in which every developer behaves the same. Therefore also the probability that a developer is among the five developers with the highest EDCPTD score should be the same for every developer. With this networks we want to test if 10.000 executions of the generation are enough for a precise result.

The following code generates two synthetic OSS projects with 25 developers. Both generated multiplex networks are weighted and undirected. The first network has only intra-layer connections. For every developer is a estimated number of 15 connections on every layer generated. The network is weighted and undirected. The second network has intra- and inter-layer connections. For every developer are half of the estimated number of connections generated as intra-layer connections and half of the estimated number of connections as inter-layer connections. In this way both networks have approximately the same number of connections in total.

```

# construct tensor only with intra-layer connections
create.equal.tensor = function(){
  array <- array(0, dim = c(25, 3, 25, 3))

  for(i in 1:25){
    for (l in 1:3) {
      for (x in 1:n.random(15)) {
        j = sample((1:25)[-i], 1)
        array[i,l,j,l] = array[i,l,j,l] +1
        array[j,l,i,l] = array[i,l,j,l]
      }
    }
  }

  tensor <- rTensor::as.tensor(array)
  return(tensor)
}
```

```
# construct tensor with intra- and inter-layer connections
create.equal.inter.tensor = function(){
  array <- array(0, dim = c(25, 3, 25, 3))

  for(i in 1:25){
    for (l in 1:3) {

      # intra-layer connections
      for (x in 1:n.random(15/2)) {
        j = sample((1:25)[-i], 1)
        array[i,l,j,l] = array[i,l,j,l] +1
        array[j,l,i,l] = array[i,l,j,l]
      }

      # inter-layer connections
      l2 = l %% 3 +1
      for (x in 1:n.random(15/2)) {
        j = sample((1:25)[-i], 1)
        array[i,l,j,l2] = array[i,l,j,l2] +1
        array[j,l2,i,l] = array[i,l,j,l2]
      }
    }
  }

  tensor <- rTensor::as.tensor(array)
  return(tensor)
}
```

As all developers behave in the same way, each developer should be considered to be among the five most important nodes with the same possibility. As the number of core developers is a fifth of the number of all developers, the estimated result should be 20%.

When this synthetic OSS projects are generated 10.000 times and EDCPTD centrality is applied on them, following results are obtained: EDCPTD centrality applied on the ordinary forth-order tensors identifies the labeled core developers in 20,0166% of the executions in the network without inter-layer connections and in 19,9756 % in the network with inter-layer connections correctly. For the EDCPTD centrality on the normalized tensor, the labeled core developers are identified in 20,0508% of the executions in the network without inter-layer connections and in 19,9406% in the network with inter-layer connections. So in all cases the estimated value is obtained quite exactly. This confirms that 10.000 is a sufficient number of repetitions.

To examine the second research question, we construct networks in four settings:

- Setting 1: core developers have more connections than peripheral developers
- Setting 2: core developers have more connections to other core developers than to peripheral developers
- Setting 3: peripheral developers have only connections to core developers
- Setting 4: peripheral developers are divided in separate clusters. Only core developers are allowed to have connections to different clusters.

5.4.2 Setting 1: Core Developers are More Active then Peripheral Developers

In the first constructed setting, we simulate the behavior that core developers are more active than peripheral developers. This includes both, communication and code contribution, so this activeness holds for all three layers. In the generated network has each labeled core developer more connections than the labeled peripheral developers. Therefore we define estimated numbers of connections for core and peripheral developers. For each developer this estimated number of connections are created on every layer. These connections are undirected, so for every entry in the tensor, also a second entry that represents the connection in the other direction is added. The following code constructs a synthetic OSS project which has approximately 20 connections on every layer for core developers, and only approximately 15 connections on every layer for peripheral developers:

```
create.more.core.connections.tensor = function(){
  array <- array(0, dim = c(25, 3, 25, 3))

  for (l in 1:3) {

    # core connections
    for(i in 1:5){
      for (x in 1:n.random(20)) {
        j = sample((1:25)[-i], 1)
        array[i,l,j,l] = array[i,l,j,l] +1
        array[j,l,i,l] = array[i,l,j,l]
      }
    }

    # peripheral connections
    for(i in 6:25){
      for (x in 1:n.random(15)) {
        j = sample((1:25)[-i], 1)
        array[i,l,j,l] = array[i,l,j,l] +1
        array[j,l,i,l] = array[i,l,j,l]
      }
    }
  }

  tensor <- rTensor::as.tensor(array)
  return(tensor)
}
```

I will execute this setting several times with different estimated number of connections. For example will be tested, if EDCPTD centrality is still able to identify a

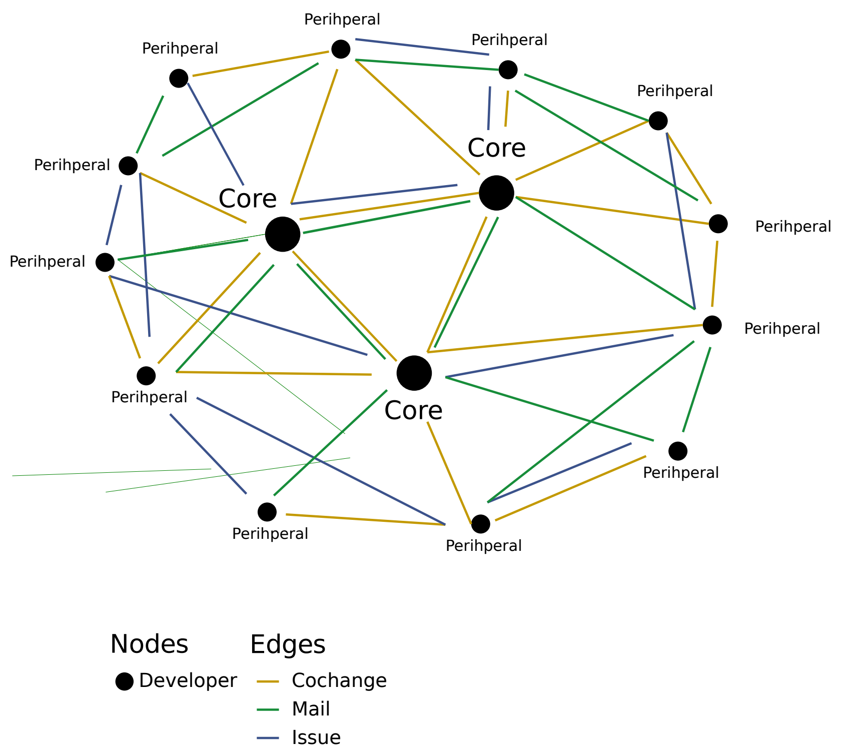

Figure 5.2: An example multiplex network for setting 1 with 15 developers and without inter-layer connections. In this OSS project are core developers more active than peripheral developers.

high rate of core developers, if the estimated number of connections for core developers is only a little higher than the estimated number of connections for peripheral developers.

5.4.3 Setting 2: Core Developers Communicate More With Other Core Developers

In the second setting, we simulate the behavior that core developers tend to communicate more with other core developers. Therefore we generate networks in which core developers have more connections to other core developers than to peripheral developers. For labeled core developers are in this networks only connections to other core developers generated. For peripheral developers are still connections to anyone generated. In this way connections between core and peripheral developers still occur, but kind of a core cluster is constructed. The code, that we use to construct a network in which core developers have approximately 15 connections to other core developers on every layer, while peripheral developers have approximately 15 connections to any developer is the following:

```
create.more.among.core.tensor = function(){
  array <- array(0, dim = c(25, 3, 25, 3))

  for (l in 1:3) {

    # core connections
    for(i in 1:5){
      for (x in 1:n.random(15)) {
        j = sample((1:5)[-i], 1)
        array[i,l,j,l] = array[i,l,j,l] +1
        array[j,l,i,l] = array[i,l,j,l]
      }
    }

    # peripheral connections
    for(i in 6:25){
      for (x in 1:n.random(15)) {
        j = sample((1:25)[-i], 1)
        array[i,l,j,l] = array[i,l,j,l] +1
        array[j,l,i,l] = array[i,l,j,l]
        }
    }

  }

  tensor <- rTensor::as.tensor(array)
  return(tensor)
}
```

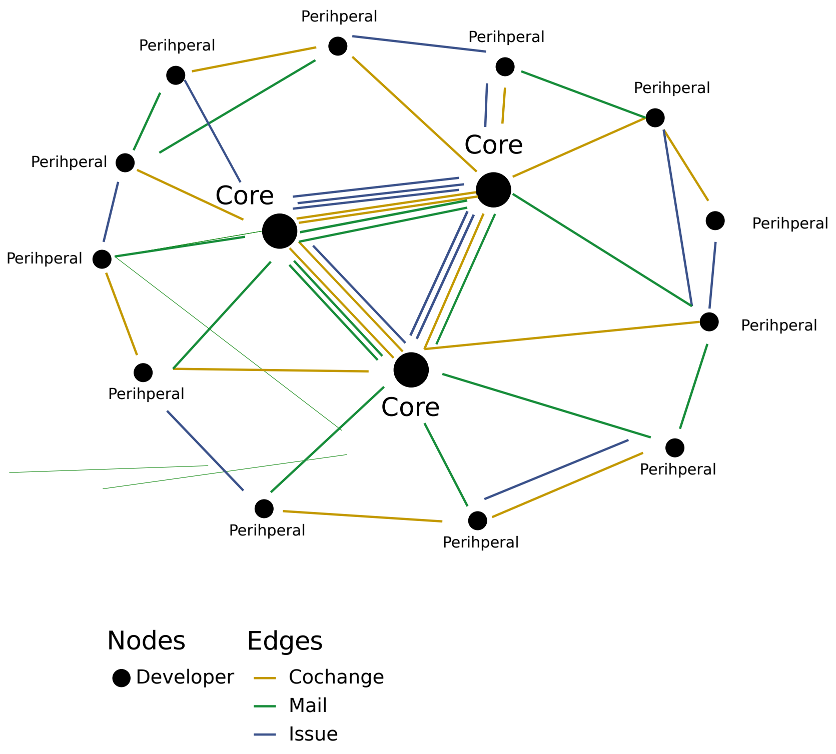

Figure 5.3: An example multiplex network for setting 2 with 15 developers and without inter-layer connections. In this OSS project have core developers more interactions with other core developers than with peripheral developers.

Again we will execute this setting with different estimated numbers of connections for core and peripheral developers. This time it could for example be interesting, if EDCPTD centrality is able to identify core developers correctly, even if they have less connections than peripheral developers.

5.4.4 Setting 3: Peripheral Developers Only Communicate With Core Developers

In the third setting, we simulate the behavior that peripheral developers not often interact with each other directly but with a core developer as middleman. Therefore we construct networks in which core developers have connections to any other developers and peripheral developers have only connections to core developers. The code, that we use to construct a network in which core developers have approximately 15 connections to any other developer on every layer, while peripheral developers have approximately 15 connections only to core developers is the following:

```r
create.middleman.core.tensor = function(){
  array <- array(0, dim = c(25, 3, 25, 3))

  for (l in 1:3) {

    # core connections
    for(i in 1:5){
      for (x in 1:n.random(15)) {
        j = sample((1:5)[-i], 1)
        array[i,l,j,l] = array[i,l,j,l] +1
        array[j,l,i,l] = array[i,l,j,l]
      }
    }

    # peripheral connections
    for(i in 6:25){
      for (x in 1:n.random(15)) {
        j = sample((1:5)[-i], 1)
        array[i,l,j,l] = array[i,l,j,l] +1
        array[j,l,i,l] = array[i,l,j,l]
      }
    }

  }

  tensor <- rTensor::as.tensor(array)
  return(tensor)
}
```

Also in this setting, we will execute several cases with different estimated numbers of connections for core and peripheral developers. Again it will be interesting if EDCPTD centrality is also able to identify a high percentage of core developers correct if the peripheral developers have more connections than the core developers.

5.4.5 Setting 4: Hierarchy Networks

The fourth constructed setting is a little bit different. In opposite to the previous settings it does not simulate the typical core-peripheral relationships that where described in Section 2.2, but a hierarchy that was discovered in some OSS projects. This hierarchy consists of several nested clusters of peripheral developers that only interact among each other. Only core developers communicate with all separate clusters. In this way core developers have a more important role than peripheral developers.

The following code simulates such a hierarchy structure. For every peripheral developer are approximately 10 connections on every layer inside his cluster generated. For every core developer are approximately five connections on every layer to other

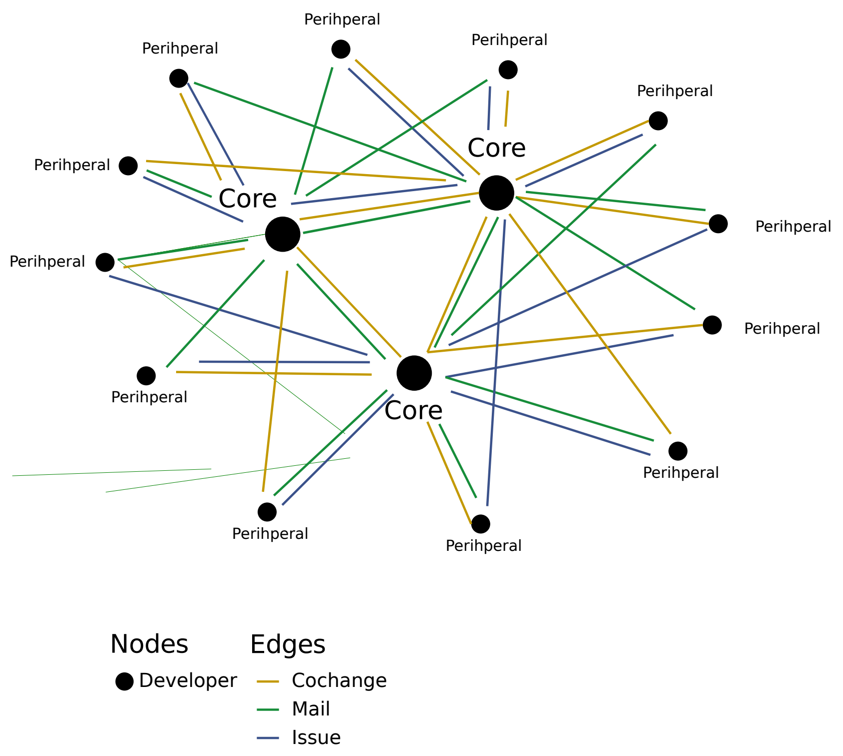

Figure 5.4: An example multiplex network for setting 3 with 15 developers and without inter-layer connections. In this OSS project interact peripheral developers only with core developers.

core developers and additionally five connections to a developers in every cluster generated. In this way core developers have also more connections than peripheral developers.

```
create.hierarchy.tensor = function(){
  array <- array(0, dim = c(25, 3, 25, 3))

  for (l in 1:3) {

    # core connections
    for(i in 1:5){

      # core group connections
      for (x in 1:n.random(5)) {
        j = sample((1:5)[-i], 1)
        array[i,l,j,l] = array[i,l,j,l] +1
        array[j,l,i,l] = array[i,l,j,l]
      }

      # core to other connections
      for (group in 2:5) {
        for (x in 1:n.random(5)) {
          j = sample((1 + (group-1)* 5) : (group * 5), 1)
          array[i,l,j,l] = array[i,l,j,l] +1
          array[j,l,i,l] = array[i,l,j,l]
        }
      }
    }

    # peripheral connections
    for (group in 2:5) {

      # connections inside a group
      for(i in (1 + (group-1)* 5) : (group * 5)){
        for(x in 1:n.random(10)){
          j = sample(((1 + (group-1)* 5) : (group * 5))[-i], 1)
          array[i,l,j,l] = array[i,l,j,l] +1
          array[j,l,i,l] = array[i,l,j,l]
        }
      }
    }

  }

  tensor <- rTensor::as.tensor(array)
  return(tensor)
}
```

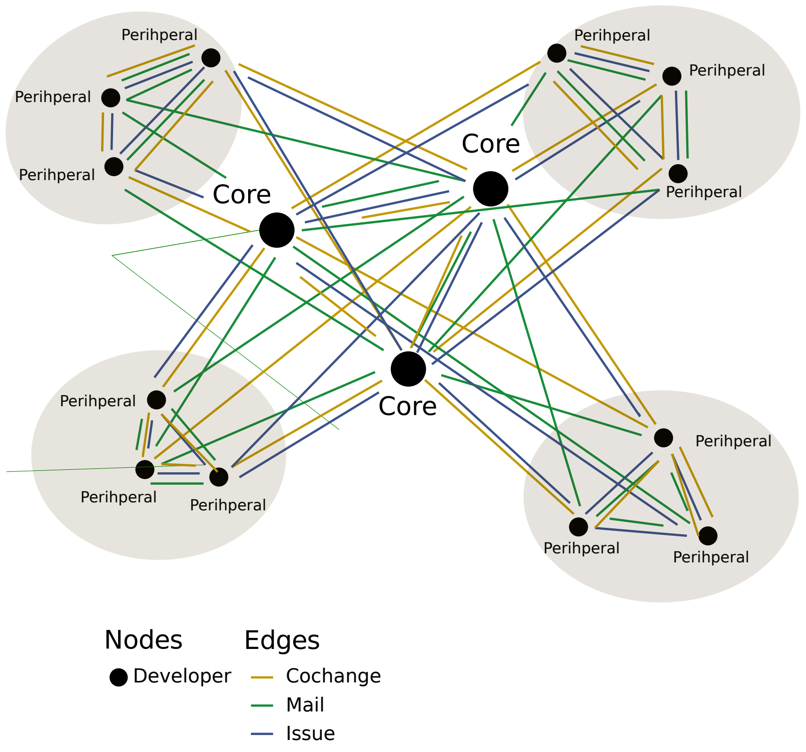

Figure 5.5: An example multiplex network for setting 4 with 15 developers and without inter-layer connections. In this OSS project are peripheral developers divided in cluster. Only core developers have connections to several clusters.

Also this setting will be executed with different numbers of connections. For example could peripheral developers communicate more among each other than core developers.

6 Evaluation

In this section we evaluate the research questions. In two sections are the results of the two research questions described and interpreted.

6.1 (RQ1) Comparison of EDCPTD centrality with Established Centrality Measures

To examine the first research question, I compared the results of EDCPTD centrality with the results of the already for the detection of core developers established centrality measures degree- and eigenvector centrality. The results of the comparisons are represented in tabular form. The tables with the results for the Apache Zeppelin data are displayed in Section 6.1 and Section 6.1. The results for the OpenSSL and OwnCloud data, which consist of more time ranges, are displayed in the appendix.

In the tables are all comparisons that were described in Section 5.3.5 represented. The first three columns contain information about the time ranges of the project that are compared. This includes the number of the range, the number of active developers in the range and how many developer belong to the top 20% of the active developers that are identified as core developers. The following columns present the results of the direct comparison of the top developers that have been identified by EDCPTD centrality applied on the multiplex network and degree- and eigenvector centrality applied on the single-layer networks. The percentage states how many of the top 20% developers of EDCPTD centrality are also among the top 20% developers of the particular one-dimensional centrality measurement. The last column indicates how many percent of the top 20% developer identified by EDCPTD centrality have also been identified to be among the top 20% of developers by any other method. In the last line the average results of the comparisons are shown.

The direct comparisons of the EDCPTD centrality with degree- and eigenvector centrality applied on the different single-layer networks show that EDCPTD centrality is able to consider the information from the cochange, the mail and the issue layer

Table 6.1: Comparison of the top 20% most important developer identified by EDCPTD centrality applied on the ordinary forth-order tensor representation of Apache Zeppelin and the top 20% developers identified by degree- and eigenvector centrality applied on the cochange, mail, issue and union single-layer networks. The results state how many percent of the most important developers that have been identified by EDCPTD centrality are also identified as one of the most important developers in the particular other centrality measurements.

Time	Active	Number	Degree Centrsality				Eigenvector Centrality				
Range	Developer	Top 20%	cochange	mail	issue	union	cochange	mail	issue	union	**Any**
1	55	11	0.55	0.82	0.18	0.82	0.55	0.82	0.18	1.00	**1.00**
2	79	16	0.19	0.69	0.19	0.69	0.19	0.88	0.25	0.88	**1.00**
3	127	26	0.38	0.73	0.35	0.54	0.42	0.65	0.38	0.77	**1.00**
4	169	34	0.59	0.85	0.53	0.79	0.59	0.59	0.56	0.82	**0.97**
Average			0.43	0.77	0.31	0.71	0.44	0.74	0.34	0.87	**0.99**

Table 6.2: Comparison of the top 20% most important developer identified by EDCPTD centrality applied on the normalized forth-order tensor representation of Apache Zeppelin and the top 20% developers identified by degree- and eigenvector centrality applied on the cochange, mail, issue and union single-layer networks. The results state how many percent of the most important developers that have been identified by EDCPTD centrality are also identified as one of the most important developers in the particular other centrality measurements.

Time	Active	Number	Degree Centrsality				Eigenvector Centrality				
Range	Developer	Top 20%	cochange	mail	issue	union	cochange	mail	issue	union	**Any**
1	55	11	0.91	0.45	0.55	0.45	1.00	0.36	0.45	0.55	**1.00**
2	79	16	0.19	0.69	0.19	0.69	0.19	0.88	0.25	0.88	**1.00**
3	127	26	0.38	0.73	0.35	0.54	0.42	0.65	0.38	0.77	**1.00**
4	169	34	0.59	0.85	0.53	0.79	0.59	0.59	0.56	0.82	**0.97**
Average			0.52	0.68	0.4	0.62	0.55	0.62	0.41	0.76	**0.99**

and combines them in one centrality score. All established centrality measurements on the single-layer networks have an accordance with EDCPTD centrality, so it can be concluded that all types of interaction have influence on the centrality score of EDCPTD centrality. In the most cases eigenvector centrality is more similar to EDCPTD centrality than degree centrality. This holds for both, EDCPTD centrality applied on the ordinary and the normalized tensor. The reason for that could be the related concept of eigenvalues and singular values that these centrality measures uses for their calculations. Further is EDCPTD centrality applied on the ordinary tensor in every project quite similar to eigenvector centrality applied on one of the single-layer networks. In Apache Zeppelin it is with an average accordance of 87% quite similar to the results of the union network, in OpenSSL with an average accordance of each 95% quite similar to the results of the cochange and the union network and in OwnCloud with an average accordance of 81% quite similar to the results of the cochange network. The results of the comparisons between EDCPTD centrality applied on the normalized tensor and the other centrality measures are closer together. So EDCPTD centrality applied on the normalized tensor, in which all layers have the same weightiness, rates all layer with a more equal importance. In this case the normalization of the tensor could make sense, as it guarantees that no layer with extraordinary much connections dominates the centrality calculations.

An even clearer result shows the rate of how many top 20% developers of EDCPTD centrality have also been identified as important by any of the other centrality measurements. In the most cases, all developer that have been identified as important by EDCPTD centrality have also in at least one of the other measurements been identified as important. The most important developers identified by EDCPTD centrality in both tensors of Apache Zeppelin, are in average 99% also identified as important by the established centrality measures. In OpenSSL this number differs a little bit for the application of EDCPTD centrality on both tensors, but with averagely 100% for the ordinary and 97% for the normalized tensors it is a high rate for both of them. In OwnCloud all developers that have been identified by EDCPTD centrality applied on the ordinary and the normalized tensors have also been identified by one of the established centrality measures. Even though the results of the ordinary tensor measurements are in some cases higher than the results of the normalized tensor measurements, all measurements fulfill the expected result. These results show, that EDCPTD centrality is able to identify core developers, that do have an important role in the OSS project.

As all established one-dimensional centrality measures have an accordance with EDCPTD centrality, it can be concluded that EDCPTD centrality is able to identify core developers that are important for the OSS project with the consideration of all interactions between developers.

6.2 (RQ2) Detection of Typical Core Developer Behavior

To test if EDCPTD centrality is able to recognize typical core behavior, I simulated different structures in several settings. All settings have different limitations about how often and in which way core and peripheral developers behave. The exact construction of the settings is described in Section 5.4.

6.2.1 Results of Setting 1

The first setting examines the behavior, that core developers are more active in communication and collaboration than peripheral developers. Therefore the estimated number of connections on each layer is higher for core developers than for peripheral developers. To rate how EDCPTD centrality is able to recognize this pattern, I executed this setting with five different estimated numbers of connections for core and peripheral developers. This includes cases in which the difference of the estimated number is high and a case in which core developers have only a few more connections than peripheral developers. To rate the results of EDCPTD centrality, I measure how many percent of the labeled core developers are identified as the most important developers with the highest EDCPTD score. The higher the rate, the better is EDCPTD centrality ale to identify this pattern of behavior.

Further to figure out how inter-layer connections affect the result of EDCPTD centrality, I simulate three different networks, one network in which only intra-layer connections for core and peripheral developers are generated, one network in which both have intra-layer connections but only for core developers are inter-layer connections to any other developer generated and one network in which core and peripheral developers have intra- and inter-layer connections. For all five cases with different estimated numbers, the three networks are generated. In order to keep the results of the different networks comparable, the estimated number of connections on a layer is split to intra- and inter-layer connections. If the developer group has inter-layer connections, half of the estimated connections point as intra-layer connections to a node on the same layer and half of the estimated connections point as inter-layer connections to a node on a different layer. In this way the total number of connections stays the same for all three networks. The results are displayed in Table 6.3

Table 6.3: The results how many percent of the labeled core developers in setting 1 have been identified by EDCPTD centrality correctly. In this setting the core developers have more connections on every layer than peripheral developers. The setting is executed with different estimated numbers of connections for core and peripheral developers on every layer.

Estimated number of connections on every layer		**No inter-layer connections**			**Core with inter-layer connections**			**All with inter-layer connections**	
		ordinary	normalized		ordinary	normalized		ordinary	normalized
30 core - 15 peripheral		0.89	0.89		1.00	1.00		1.00	1.00
25 core - 15 peripheral		0.74	0.74		0.94	0.94		0.94	1.00
20 core - 15 peripheral		0.49	0.49		0.66	0.67		0.66	0.66
20 core - 10 peripheral		0.79	0.79		0.98	0.98		0.97	0.97
20 core - 5 peripheral		0.94	0.94		1.00	1.00		1.00	1.00

The results of the networks without inter-layer connections show that the bigger the difference between the estimated number of connections for core and peripheral developers is, the higher is the rate of correct identified core developers. For example in the case in which peripheral developers have only five connections, 94% of the core developers are identified correctly. This result indicate that EDCPTD centrality can differentiate if a developer is regularly or just temporary active in the OSS project. A small difference of the estimated number of connections of core and peripheral developers is still noticed in 49% of the cases correctly, which is still almost in the half of the executions, but could be higher. The higher the proportional difference of estimated connections gets, the higher gets the rate. But however, EDCPTD centrality is in no case able to identify all core developers correctly.

The results of the networks with inter-layer connections only for core developers and the results of the networks with inter-layer connections for all developers are approximately the same. Again, the bigger the difference between the estimated numbers is, the more core developers are identified correctly. But the rate of correct identified core developers is in all cases higher than in the networks without inter-layer connections. In the cases with a big difference of the estimated numbers of connections, even all core developers are identified correctly. Apparently is the additional influence of the inter-layer connections more important for the identification of core developer than just the number of connections.

Further it is noticeable, that there is almost no difference in the results if EDCPTD centrality is applied on the ordinary or the normalized tensor.

6.2.2 Results of Setting 2

In the second setting, the behavior that core developers interact more with other core developers is simulated. Therefore only connections to other core developers are generated for the labeled core developers, while connections to anyone in the project are generated for peripheral developers. In this way, a network in which core developers are arranged in a central cluster is simulated. To differentiate this behavior from the behavior of the first setting, in which core developer haven been more active than peripheral developers, I examine cases in which core developers have a lower or equal estimated number of connections than peripheral developers. The higher the rate of correct identified core developer, the better is EDCPTD centrality able to identify this core behavior.

Again, these cases are executed three times, without inter-layer connections, with inter-layer connections only for core developers and with inter-layer connections for both developer groups. If a developer group has inter-layer connections, half of the estimated connections are generated as intra-layer connections and half of the estimated connections are generated as inter-layer connections. The inter-layer connections are generated with the same behavior as intra-layer connections, so core developers have only inter-layer connections to other core developers. The results of this setting are displayed in Table 6.4.

Table 6.4: The results how many percent of the labeled core developers in setting 2 have been identified by EDCPTD centrality correctly. In this setting core developers have more connections to other core developers than to peripheral developers. The setting is executed with different estimated numbers of connections for core and peripheral developers on every layer.

Estimated number of connections on every layer		**No inter-layer connections**			**Core with inter-layer connections**			**All with inter-layer connections**	
		ordinary	normalized		ordinary	normalized		ordinary	normalized
15 core - 25 peripheral		0.80	0.81		0.93	1.00		0.94	0.93
15 core - 20 peripheral		1.00	1.00		0.99	1.00		1.00	1.00
15 core - 15 peripheral		1.00	1.00		1.00	1.00		1.00	1.00
10 core - 15 peripheral		0.86	0.87		0.95	1.00		0.97	0.96
5 core - 15 peripheral		0.03	0.03		0.01	1.00		0.02	0.02

In the network without inter-layer connections, EDCPTD centrality is in the most cases able to identify a high rate of core developers correctly. Especially in the case in which the estimated number of connections for core and peripheral developers is the same, all core developers are identified correctly. Also in the cases in which peripheral developer have a few more connections than core developers, the most core developers are still identified correct. This indicates that EDCPTD centrality is able to detect the behavior that core developer in OSS projects tend to interact more among each other. Further, as the rates in this result are higher than the rates of setting 1, it can be implicated that the centrality of a node in the network structure is more important for EDCPTD centrality than just the number of connections of a node. Only in the case in which core developers have approximately five connections on every layer, while peripheral developers have approximately fifteen connections on every layer, the core developers are just in a low rate identified correctly. A reason for this could be that due the small number of core connections, the pattern that core developers communicate more with other core developers is not noticeable. As in real OSS projects core developers usually are more active, this result does not play an important role for the application on real OSS data.

The rates for EDCPTD centrality applied on the ordinary tensor of the networks in which only core developers have inter-layer connections are quite similar, or even a little higher. The results for the normalized tensor of these networks are different. Surprisingly, all core developers have been identified correctly. Even in the case in which barely any correct core developer in the other networks is identified. A reason for this could be, that due the setting only inter-layer connections from core to core developers are created. So peripheral developers do not occur in the inter-layer connections. While these inter-layer connections are just not rated as that important in the ordinary tensor, they are considered with the same weightiness as layers in the normalized tensor. Thereby the not-occurrence of peripheral developers in inter-layer connections has a bigger influence on the result.

In the networks in which inter-layer connections for both developer groups are generated, the results are again equal for the ordinary and normalized tensor. In most cases the core developers are identified correctly. Even with a better rate than in the networks without inter-layer connections.

6.2.3 Results of Setting 3

The third setting simulates the behavior, that peripheral developers do not often communicate directly with each other, but with a core developer as middleman. Therefore are for peripheral developers only connections to core developers generated. For core developers connections to any other developer are generated. Similar to the second setting, I analyze cases in which core developer have less or equal connections than peripheral developers, to distinguish this setting from the first setting.

Also for these cases three different networks are generated, one without inter-layer connections, one with inter-layer connections only for core developers and one with inter-layer connections for all developers. If inter-layer connections exist, half of the estimated number of connections on every layer are generated as intra-layer connections and half of the estimated number of connections are generated as inter-layer connections. So the total number of connections for core and peripheral developers stays approximately the same. Also the limitations for inter-layer connections are the same as for intra-layer connections. This means, that peripheral developer only have inter-layer connections to core developers. The results of this setting are represented in Table 6.5.

Table 6.5: The results how many percent of the labeled core developers in setting 3 have been identified by EDCPTD centrality correctly. In this setting peripheral developers have only connections to core developers. The setting is executed with different estimated numbers of connections for core and peripheral developers on every layer.

Estimated number of connections on every layer		**No inter-layer connections**			**Core with inter-layer connections**			**All with inter-layer connections**	
		ordinary	normalized		ordinary	normalized		ordinary	normalized
5 core - 15 peripheral		1.00	1.00		1.00	0.64		1.00	1.00
10 core - 15 peripheral		1.00	1.00		1.00	0.93		1.00	1.00
15 core - 15 peripheral		1.00	1.00		1.00	1.00		1.00	1.00
15 core - 30 peripheral		1.00	1.00		1.00	1.00		1.00	1.00
15 core - 45 peripheral		1.00	1.00		1.00	1.00		1.00	1.00

The third setting has the most definite results. In the networks without inter-layer connections and in the networks in which core and peripheral developers have inter-layer connections, all core developers are identified correctly. This holds for the case with equal estimated connections for core and peripheral developers and even in the cases in which peripheral developers have more connections than core developers. It can be succeeded, that EDCPTD centrality is well able to recognize this pattern and identify all developers correctly. Again, this indicates that central network structures have more influence on the result of EDCPTD centrality than just the number of connections.

The networks in which only core developers have inter-layer connections do again show irregularities. While in the ordinary tensors all core developers have been identified correctly, EDCPTD centrality applied on the normalized tensor has a few problems to identify all core developers in the cases with only a few core connections. A reason for this could be, as only peripheral developers have a limitation in their behavior, the wanted pattern is not represented on the inter-layer connections. With the normalization of this tensor, the few inter-layer connections which do not represent the wanted pattern of behavior, become more important for the result. As soon as the number of core developer rises, also in this networks are all core developers identified correctly. So again, as core developers usually have many connections, this result is not that important for the application on real OSS data.

Because the core developers are, apart from the edge cases that do not model the setting, always identified correctly, it can be assumed that EDCPTD centrality can perfectly identify this core developer behavior in OSS projects.

6.2.4 Results of Setting 4

The forth setting simulates a hierarchy cluster structure in OSS projects. Therefore the peripheral developer are divided in four different clusters. Each peripheral developer has only connections to a developer in his cluster. Core developers have connections to any developer. To evaluate this setting, cases with three different estimated numbers of connections on every layer are executed. The first estimated number defines how many connections core developer have to other core developer, the second estimated number defines how many connections a core developer has to each of the four peripheral clusters and the third estimated number defines how many connections a peripheral developer has inside his cluster.

As in the previous three settings, for all cases are networks only with intra-layer connections, networks with inter-layer connections only for core developers and networks with inter-layer connections for all developers generated. The estimated number of connections on every layer is split in intra- and inter-layer connections and inter-layer connections behave the same as intra-layer connections. In this way the proportion of the three different networks stays the same for every case.

As in this setting the estimated number of connections between core developers, the estimated number of connections from core to peripheral developers and the estimated number of connections for peripheral developers inside their cluster can be changed, a bigger number of different cases can be evaluated. The most interesting results are displayed in Table 6.6

Table 6.6: The results how many percent of the labeled core developers in setting 4 have been identified by EDCPTD centrality correctly. In this setting peripheral developers are divided in four cluster and only interact with peripheral developers in the same cluster. Core developer interact with each other and with every peripheral cluster. The setting is executed with different estimated numbers of connections between core developers, between core developers and peripheral developers in every clusters, and between peripheral developers inside their clusters.

Estimated number of connections on every layer		**No inter-layer connections**			**Core with inter-layer connections**			**All with inter-layer connections**	
		ordinary	normalized		ordinary	normalized		ordinary	normalized
5 core to core 5 core to peripheral 20 in peripheral cluster		0.00	0.00		0.00	1.00		0.00	0.00
20 core to core 5 core to peripheral 20 in peripheral cluster		1.00	1.00		0.99	1.00		1.00	0.99
20 core to core 5 core to peripheral 10 in peripheral cluster		1.00	1.00		1.00	1.00		1.00	1.00
5 core to core 10 core to peripheral 5 in peripheral cluster		1.00	1.00		1.00	1.00		1.00	1.00

The results of the different cases indicate different conclusions. In the first case is, apart of the normalized network with only inter-layer connections for core developers, no core developer identified correctly. In this case core developer have approximately five connection on every layer to other core developer and another five connections to every peripheral cluster. So approximately 25 connections on every layer in total. Peripheral developers have 20 connections to other peripheral developers inside their cluster on every layer. Additionally they are involved in approximately five connections from core developers. So core and peripheral developers have more or less the same number of connections. That in this case no core developer is identified correctly, indicates that the hierarchy cluster structure alone is not sufficient for EDCPTD centrality to detect core developers.

In the other cases are core developers identified correctly in a high rate. In these cases is the hierarchy cluster structure combined with the typical core behaviors of the previous settings. In the second case have core developers more connections than peripheral developers. Here the core developers are identified in the most cases correctly. In the third case have core developers more connections to other core developers. Because of the limitation that peripheral developers only communicate inside their cluster, this case cannot be separated from the case that core developers also have more connections than peripheral developers. In this case all core developers are correctly identified. The forth case combines the hierarchical clusters with the behavior that peripheral developers have more connections to core developers than to other peripheral developers. Again all core developers are identified correctly.

Only the normalized networks in which just core developers have inter-layer connections have a different result. In this networks all core developers are identified correctly. The reason for this could be that in this networks no inter-layer connections without a core developer are generated. So the peripheral cluster do not occur. While this inter-layer connections are not considered as important in the ordinary tensor, the normalization makes them as important as normal layers.

In conclusion EDCPTD centrality is not able to identify the hierarchy cluster structure without the other characteristics of core developers. But as soon as this structure occurs in an OSS project in which also at least on of typical patterns for core developers that where examined in the previous settings occurs, EDCPTD centrality is able to identify most core developers correctly.

7 Conclusion

In this thesis I examined how useful multiplex centrality measures are to identify core developers in OSS projects. In particular I analyzed the results of the multiplex centrality measurement EDCPTD centrality applied on OSS projects. As EDCPTD centrality is applied on a multiplex network, it is possible to consider the influence of different types of interactions in the OSS project on the centrality calculation.

In the first research question I compared the results of EDCPTD centrality applied on real OSS projects with the results of degree- and eigenvector centrality, which are approved centrality measures to identify important developers in OSS projects, applied on the same OSS projects. This comparison confirmed that EDCPTD centrality is able to identify developers that are important in real OSS projects. Almost all developers that are identified by EDCPTD centrality to be among the most important developers in the project have also been identified as important by one of the established centrality measures. As EDCPTD centrality is, different to degree- and eigenvector centrality, applied on a multiplex network, it is able to consider all different types of interactions from OSS projects in its calculation. Thereby EDCPTD centrality can also recognize influences between different types of interaction and has in this way an advantage over the established centrality measures.

In the second research question I tested if EDCPTD centrality is able to identify developers that fulfill typical core developer behavior in synthetic OSS projects. In most cases, EDCPTD centrality identified a high rate of the labeled core developers correctly. Especially in the setting in which core developers interact more with other core developers and in the setting in which peripheral developers interact more with core developers, most core developers are identified correct. Even in the cases in which core developer have less connections than peripheral developers EDCPTD centrality was able to identify a high rate of core developers. The high rate of correctly identified developers in these settings indicates that the network structure has more influence on the result of EDCPTD centrality than just the number of connections of a node.

Further are the rates of correctly identified developers in the networks in which core and peripheral developers have intra- and inter-layer connections in almost all cases higher or equal than the rates in the networks in which both developer groups have only intra-layer connections. So the use of inter-layer connections in an OSS project has positive influence on the identification of core developers. The introduction of inter-layer connections, could therefore help to model OSS projects more accurate and enable a more precise centrality measurement.

In the first research question have the results of the normalized tensor less difference from each other than the results of the ordinary tensor. In this case the normalization insured that no layer is weighted with a too high importance. In the second research question the results for the ordinary and the normalized tensor are in many cases the same. This is not surprising as all layers have approximately the same number of connections. Only in the networks in which only core developer has inter-layer connections, the normalized tensor caused in some cases a unexpected result. This is an example that the normalization of the tensors can sometimes also make unimportant layers more important. After all, cases in which the normalization leads to a better result as well as cases in which the normalization can falsify the result exist.

In conclusion, EDCPTD is well able to identify core developers in OSS projects. The developers with the highest EDCPTD centrality score are confirmed important developers in the OSS project. Developers that fulfill typical core behavior have a higher EDCPTD centrality score than others.

8 Future Work

As in all synthetically constructed networks the result of the networks in which both developer groups have inter-layer connections are better than the results of the networks without inter-layer connections, inter-layer connections in real OSS projects may have a positive influence on the identification of core developers. Therefore it is reasonable to create inter-layer connections in the retrieval of real OSS project data. For example could a reviewer on the issue layer could be with the corresponding committer on the cochange layer. Or it could be analyzed if a mail or issue text includes a hash code of a commit, so that the author of the mail/issue can be linked via an inter-layer connection with the committer. The comparison of RQ1 could than be executed again with inter-layer connections for EDCPTD centrality and an even more accurate centrality measurement can be expected.

Further both research questions showed that EDCPTD centrality is able to order developers according their importance and identify a fixed amount of most important developers correctly. But as the exact number of core developers in a OSS project is not known, it would be interesting to find out if EDCPTD centrality is also able to figure out how many core developer an OSS project has. This could for example work if EDCPTD centrality realizes a high difference between the EDCPTD scores of two successive ordered developers. All developers whose score is above this scores could be identified as core developers and all developers whose score is below this score could be identified as peripheral developers. Therefore an execution of the setting from RQ2 with a varying number of core developers would be interesting. In this settings it could be tested if EDCPTD centrality is able to identify all labeled core developers, even though it is not know how many labeled core developers exist.

A Appendix

Table A.1: Comparison of the top 20% most important developer identified by EDCPTD centrality applied on the ordinary forth-order tensor representation of Apache Zeppelin and the top 20% developers identified by degree- and eigenvector centrality applied on the cochange, mail, issue and union single-layer networks. The results state how many percent of the most important developers that have been identified by EDCPTD centrality are also identified as one of the most important developers in the particular other centrality measurements.

Time	Active	Number	Degree Centrsality				Eigenvector Centrality				
Range	Developer	Top 20%	cochange	mail	issue	union	cochange	mail	issue	union	**Any**
1	55	11	0.55	0.82	0.18	0.82	0.55	0.82	0.18	1.00	**1.00**
2	79	16	0.19	0.69	0.19	0.69	0.19	0.88	0.25	0.88	**1.00**
3	127	26	0.38	0.73	0.35	0.54	0.42	0.65	0.38	0.77	**1.00**
4	169	34	0.59	0.85	0.53	0.79	0.59	0.59	0.56	0.82	**0.97**
Average			0.43	0.77	0.31	0.71	0.44	0.74	0.34	0.87	**0.99**

Table A.2: Comparison of the top 20% most important developer identified by EDCPTD centrality applied on the normalized forth-order tensor representation of Apache Zeppelin and the top 20% developers identified by degree- and eigenvector centrality applied on the cochange, mail, issue and union single-layer networks. The results state how many percent of the most important developers that have been identified by EDCPTD centrality are also identified as one of the most important developers in the particular other centrality measurements.

Time	Active	Number	Degree Centrsality				Eigenvector Centrality				
Range	Developer	Top 20%	cochange	mail	issue	union	cochange	mail	issue	union	**Any**
1	55	11	0.91	0.45	0.55	0.45	1.00	0.36	0.45	0.55	**1.00**
2	79	16	0.19	0.69	0.19	0.69	0.19	0.88	0.25	0.88	**1.00**
3	127	26	0.38	0.73	0.35	0.54	0.42	0.65	0.38	0.77	**1.00**
4	169	34	0.59	0.85	0.53	0.79	0.59	0.59	0.56	0.82	**0.97**
Average			0.52	0.68	0.4	0.62	0.55	0.62	0.41	0.76	**0.99**

Table A.3: Comparison of the top 20% most important developer identified by EDCPTD centrality applied on the normalized forth-order tensor representation of OpenSSL and the top 20% developers identified by degree- and eigenvector centrality applied on the cochange, mail, issue and union single-layer networks. The results state how many percent of the most important developers that have been identified by EDCPTD centrality are also identified as one of the most important developers in the particular other centrality measurements.

Time	Active	Number	Degree Centrsality				Eigenvector Centrality				
Window	Developer	Top 20%	cochange	mail	issue	union	cochange	mail	issue	union	**Overall**
1	11	3	1.00	0.00	0.33	1.00	1.00	0.33	0.33	1.00	**1.00**
2	14	3	0.67	0.00	0.67	0.67	1.00	0.00	0.67	1.00	**1.00**
3	11	3	1.00	0.33	0.33	1.00	1.00	0.33	0.33	1.00	**1.00**
4	13	3	1.00	0.33	0.33	1.00	1.00	0.33	0.33	1.00	**1.00**
5	13	3	0.33	0.67	0.33	0.67	0.33	1.00	0.67	0.33	**1.00**
6	12	3	1.00	0.67	0.33	1.00	1.00	0.67	0.33	1.00	**1.00**
7	14	3	1.00	0.00	0.33	1.00	1.00	0.33	0.33	1.00	**1.00**
8	15	3	0.67	0.33	0.33	0.67	1.00	0.33	0.67	1.00	**1.00**
9	13	3	1.00	0.33	0.33	1.00	1.00	0.67	0.33	1.00	**1.00**
10	19	4	1.00	0.25	0.00	1.00	1.00	0.25	0.25	1.00	**1.00**
11	18	4	0.75	0.00	0.25	0.75	1.00	0.00	0.5	1.00	**1.00**
12	18	4	1.00	0.25	0.5	1.00	1.00	0.5	0.5	1.00	**1.00**
13	20	4	0.75	0.00	0.5	0.75	1.00	0.25	0.5	1.00	**1.00**
14	21	5	0.8	0.4	0.4	0.8	1.00	0.2	0.6	1.00	**1.00**
15	21	5	1.00	0.6	0.6	1.00	1.00	0.4	0.8	1.00	**1.00**
16	23	5	0.8	0.4	0.4	0.8	1.00	0.2	0.8	1.00	**1.00**
17	21	5	1.00	0.4	0.6	1.00	1.00	0.4	0.8	1.00	**1.00**
18	24	5	1.00	0.6	0.6	1.00	1.00	0.6	1.00	1.00	**1.00**
19	22	5	0.8	0.2	0.4	0.8	1.00	0.4	0.8	1.00	**1.00**
20	27	6	1.00	0.33	0.5	1.00	1.00	0.5	0.67	1.00	**1.00**
21	28	6	1.00	0.33	0.5	1.00	1.00	0.33	0.67	1.00	**1.00**
22	32	7	1.00	0.29	0.43	1.00	1.00	0.29	0.71	1.00	**1.00**
23	36	8	0.75	0.5	0.62	0.75	1.00	0.62	0.25	1.00	**1.00**
24	48	10	1.00	0.4	0.3	1.00	1.00	0.4	0.6	1.00	**1.00**
25	121	25	0.96	0.4	0.32	0.88	1.00	0.4	0.24	1.00	**1.00**
26	126	26	0.96	0.38	0.31	0.88	1.00	0.38	0.5	0.96	**1.00**
27	150	30	1.00	0.43	0.37	0.97	0.97	0.4	0.37	1.00	**1.00**
28	178	36	0.97	0.39	0.47	0.83	1.00	0.36	0.53	0.89	**1.00**
29	206	42	0.98	0.33	0.45	0.86	0.98	0.38	0.43	1.00	**1.00**
30	207	42	0.93	0.4	0.55	0.9	0.98	0.45	0.55	0.98	**1.00**
31	241	49	0.92	0.41	0.51	0.84	0.98	0.39	0.53	0.88	**1.00**
32	242	49	0.96	0.41	0.49	0.82	0.96	0.67	0.49	0.82	**1.00**
33	248	50	0.5	0.3	0.94	0.7	0.44	0.56	0.98	0.68	**1.00**
34	322	65	0.55	0.23	0.97	0.68	0.52	0.65	0.98	0.65	**1.00**
Average			0.88	0.33	0.45	0.88	0.95	0.41	0.56	0.95	**1**

Table A.4: Comparison of the top 20% most important developer identified by EDCPTD centrality applied on the normalized forth-order tensor representation of OpenSSL and the top 20% developers identified by degree- and eigenvector centrality applied on the cochange, mail, issue and union single-layer networks. The results state how many percent of the most important developers that have been identified by EDCPTD centrality are also identified as one of the most important developers in the particular other centrality measurements.

Time	Active	Number	Degree Centrsality				Eigenvector Centrality				
Window	Developer	Top 20%	cochange	mail	issue	union	cochange	mail	issue	union	**Any**
1	11	3	1.00	0.00	0.33	1.00	1.00	0.33	0.33	1.00	**1.00**
2	14	3	0.00	0.67	0.33	0.00	0.00	1.00	0.33	0.00	**1.00**
3	11	3	0.33	0.67	0.33	0.33	0.33	0.67	0.00	0.33	**0.67**
4	13	3	0.33	1.00	0.00	0.33	0.33	1.00	0.33	0.33	**1.00**
5	13	3	0.67	0.33	0.33	0.67	0.67	0.33	0.67	1.00	**1.00**
6	12	3	0.67	1.00	0.33	0.67	0.67	1.00	0.33	0.67	**1.00**
7	14	3	0.33	0.67	0.33	0.33	0.33	1.00	0.33	0.33	**1.00**
8	15	3	0.33	1.00	0.33	0.33	0.33	1.00	0.33	0.33	**1.00**
9	13	3	1.00	0.33	0.33	1.00	1.00	0.67	0.33	1.00	**1.00**
10	19	4	0.25	1.00	0.5	0.25	0.25	1.00	0.25	0.25	**1.00**
11	18	4	0.75	0.00	0.25	0.75	1.00	0.00	0.5	1.00	**1.00**
12	18	4	1.00	0.25	0.5	1.00	1.00	0.5	0.5	1.00	**1.00**
13	20	4	0.25	0.75	0.25	0.25	0.00	0.75	0.5	0.00	**0.75**
14	21	5	0.8	0.4	0.4	0.8	1.00	0.2	0.6	1.00	**1.00**
15	21	5	1.00	0.6	0.6	1.00	1.00	0.4	0.8	1.00	**1.00**
16	23	5	0.8	0.4	0.4	0.8	1.00	0.2	0.8	1.00	**1.00**
17	21	5	0.4	0.8	0.4	0.4	0.4	1.00	0.6	0.4	**1.00**
18	24	5	0.6	0.8	0.2	0.6	0.6	1.00	0.6	0.6	**1.00**
19	22	5	0.8	0.2	0.4	0.8	1.00	0.4	0.8	1.00	**1.00**
20	27	6	1.00	0.33	0.5	1.00	1.00	0.5	0.67	1.00	**1.00**
21	28	6	1.00	0.33	0.5	1.00	1.00	0.33	0.67	1.00	**1.00**
22	32	7	1.00	0.29	0.43	1.00	1.00	0.29	0.71	1.00	**1.00**
23	36	8	0.38	0.75	0.38	0.38	0.5	0.75	0.25	0.5	**0.75**
24	48	10	1.00	0.4	0.3	1.00	1.00	0.4	0.6	1.00	**1.00**
25	121	25	0.36	0.76	0.48	0.48	0.36	0.72	0.32	0.36	**0.92**
26	126	26	0.62	0.5	0.54	0.69	0.62	0.54	0.31	0.65	**1.00**
27	150	30	1.00	0.43	0.37	0.97	0.97	0.4	0.37	1.00	**1.00**
28	178	36	0.97	0.39	0.47	0.83	1.00	0.36	0.53	0.89	**1.00**
29	206	42	0.43	0.45	0.9	0.55	0.43	0.45	0.98	0.43	**0.98**
30	207	42	0.93	0.4	0.55	0.9	0.98	0.45	0.55	0.98	**1.00**
31	241	49	0.92	0.41	0.51	0.84	0.98	0.39	0.53	0.88	**1.00**
32	242	49	0.96	0.41	0.49	0.82	0.96	0.67	0.49	0.82	**1.00**
33	248	50	0.5	0.3	0.94	0.7	0.44	0.56	0.98	0.68	**1.00**
34	322	65	0.55	0.23	0.97	0.68	0.52	0.65	0.98	0.65	**1.00**
Average			0.67	0.51	0.44	0.68	0.7	0.59	0.53	0.71	**0.97**

Table A.5: Comparison of the top 20% most important developer identified by EDCPTD centrality applied on the ordinary forth-order tensor representation of OwnCloud and the top 20% developers identified by degree- and eigenvector centrality applied on the cochange, mail, issue and union single-layer networks. The results state how many percent of the most important developers that have been identified by EDCPTD centrality are also identified as one of the most important developers in the particular other centrality measurements.

Time	Active	Number	Degree Centrsality				Eigenvector Centrality				
Window	Developer	Top 20%	cochange	mail	issue	union	cochange	mail	issue	union	**Any**
1	61	13	0.85	0.46	0.69	0.77	1.00	0.38	0.69	0.85	**1.00**
2	110	22	0.86	0.59	0.5	0.77	1.00	0.59	0.5	0.73	**1.00**
3	141	29	0.93	0.48	0.52	0.72	1.00	0.48	0.55	0.72	**1.00**
4	172	35	0.94	0.29	0.49	0.77	1.00	0.34	0.54	0.8	**1.00**
5	230	46	0.96	0.28	0.57	0.78	1.00	0.35	0.54	0.76	**1.00**
6	252	51	0.47	0.35	0.92	0.71	0.51	0.53	0.98	0.63	**1.00**
7	253	51	0.78	0.24	0.53	0.71	0.92	0.76	0.51	0.82	**1.00**
8	298	60	0.45	0.27	0.9	0.63	0.5	0.5	0.97	0.6	**1.00**
9	246	50	0.88	0.2	0.5	0.7	1.00	0.52	0.5	0.78	**1.00**
10	243	49	0.61	0.29	0.94	0.73	0.55	0.49	1.00	0.65	**1.00**
11	236	48	0.54	0.23	0.96	0.73	0.48	0.44	1.00	0.73	**1.00**
12	240	48	0.94	0.19	0.52	0.9	1.00	0.56	0.52	0.85	**1.00**
13	258	52	0.62	0.23	0.96	0.71	0.6	0.5	0.9	0.69	**1.00**
Average			0.76	0.32	0.69	0.74	0.81	0.5	0.71	0.74	**1**

Table A.6: Comparison of the top 20% most important developer identified by EDCPTD centrality applied on the normalized forth-order tensor representation of OwnCloud and the top 20% developers identified by degree- and eigenvector centrality applied on the cochange, mail, issue and union single-layer networks. The results state how many percent of the most important developers that have been identified by EDCPTD centrality are also identified as one of the most important developers in the particular other centrality measurements.

Time	Active	Number	Degree Centrsality				Eigenvector Centrality				
Window	Developer	Top 20%	cochange	mail	issue	union	cochange	mail	issue	union	**Any**
1	61	13	0.85	0.46	0.69	0.77	1.00	0.38	0.69	0.85	**1.00**
2	110	22	0.86	0.59	0.5	0.77	1.00	0.59	0.5	0.73	**1.00**
3	141	29	0.93	0.48	0.52	0.72	1.00	0.48	0.55	0.72	**1.00**
4	172	35	0.94	0.29	0.49	0.77	1.00	0.34	0.54	0.8	**1.00**
5	230	46	0.96	0.28	0.57	0.78	1.00	0.35	0.54	0.76	**1.00**
6	252	51	0.43	0.37	0.88	0.67	0.47	0.55	0.92	0.61	**1.00**
7	253	51	0.51	0.31	0.84	0.76	0.49	0.47	0.96	0.71	**1.00**
8	298	60	0.48	0.27	0.78	0.67	0.53	0.47	0.72	0.65	**0.95**
9	246	50	0.5	0.3	0.88	0.68	0.5	0.42	1.00	0.64	**1.00**
10	243	49	0.61	0.29	0.94	0.73	0.55	0.49	1.00	0.65	**1.00**
11	236	48	0.54	0.23	0.96	0.73	0.48	0.44	1.00	0.73	**1.00**
12	240	48	0.56	0.29	0.96	0.62	0.54	0.4	0.98	0.65	**1.00**
13	258	52	0.62	0.23	0.96	0.71	0.6	0.5	0.9	0.69	**1.00**
Average			0.68	0.34	0.77	0.72	0.7	0.45	0.79	0.71	**1**

Bibliography

[Abe07] Mark Aberdour. Achieving Quality in Open-Source Software. *IEEE software*, 24(1):59–64, 2007. (cited on Page 6 and 7)

[Bon07] Phillip Bonacich. Some unique properties of eigenvector centrality. *Social Networks*, 29(4):pp 555–564, 2007. (cited on Page 13)

[CH03] Kevin Crowston and James Howison. The Social Structure of Open Source Software Development Teams. *School of Information Studies - Faculty Scholarship*, 2003. (cited on Page 6, 7, and 8)

[CS17] Kevin Crowston and Ivan Shamshurin. Core-periphery communication and the success of free/libre open source software projects. *Journal of Internet Servicesand Applications*, 8(10), 2017. (cited on Page 1 and 7)

[DWZ17] Haitao Wang Dingjie Wang and Xiufen Zou. Identifying key nodes in multilayer networks based on tensor decomposition. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 26(6), 2017. (cited on Page 15, 16, 17, and 18)

[FG07] Chaim Fershtman and Neil Gandal. Open source software: Motivation and restrictive licensing. *International Economics and Economic Policy*, 4(2):pp 209–225, 2007. (cited on Page 1 and 5)

[Fis89] Gerd Fischer. *Lineare Algebra.* vieweg studium, The address, 9 edition, 1989. (cited on Page 13)

[GM02] Renee Tynan Gregory Madey, Vincent Freeh. THE OPEN SOURCE SOFTWARE DEVELOPMENT PHENOMENON: AN ANALYSIS BASED ON SOCIAL NETWORK THEORY. *AMCIS 2002 Proceedings*, 2002. (cited on Page 5)

[Gos] Alexej Gossmann. Understanding the candecomp/parafac tensor decomposition. https://www.alexejgossmann.com/tensor_decomposition_CP/. Accessed: 2019-08-15. (cited on Page 17)

[GR] Eric W. Weisstein Gordon Royle. Mathworld–a wolfram web resource: "reducible matrix". http://mathworld.wolfram.com/ReducibleMatrix.html. Accessed: 2019-08-15. (cited on Page 14)

[JAHM17] Mitchell Joblin, Sven Apel, Claus Hunsen, and Wolfgang Mauerer. Classifying Developers into Core and Peripheral: An Empirical Study on

Count and Network Metrics. *International Conference on Software Engineering*, 39, 2017. (cited on Page 7, 9, and 12)

[JAM17] Mitchell Joblin, Sven Apel, and Wolfgang Mauerer. Evolutionary trends of developer coordination: a network approach. *Empirical Software Engineering*, 22(4):pp 2050–2094, 2017. (cited on Page 9)

[Joh] Prof. Dr. Volker John. Numerik i. https://www.wias-berlin.de/people/john/LEHRE/NUMERIK_I/numerik_1_1.pdf. Accessed: 2019-08-15. (cited on Page 17)

[KCH06] Qing Li Kevin Crowston, Kangning Wei and James Howison. Core and periphery in Free/Libre and Open Source software team communications. *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS'06)*, 6:pp. 118a–118, 2006. (cited on Page 2)

[Kri06] Sandeep Krishnamurthy. On the intrinsic and extrinsic motivation of free/libre/open source (FLOSS) developers. *Knowledge, Technology & Policy*, 4(18):pp 17–39, 2006. (cited on Page 5)

[Lui13] Luis Sola, Miguel Romance, Regino Criado, Julio Flores, Alejandro García del Amo and Stefano Boccaletti. Eigenvector centrality of nodes in multiplex networks . *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 23(3), 2013. (cited on Page 12)

[MRP+14] G Giulia Menichetti, Daniel Remondini, Pietro Panzarasa, Raúl J. Mondragón, and Ginestra Bianconi. Weighted Multiplex Networks. 2014. (cited on Page 15)

[NYN+02] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. Evolution patterns of open-source software systems and communities. *Proceedings of the International Workshop on Principles of Software Evolution*, pages Pages 76–85, 2002. (cited on Page 6)

[PG06] Stephen P.Borgatti and Martin G.Everett. A Graph-theoretic perspective on centrality. *Social Networks*, 28(4):pp 466–484, 2006. (cited on Page 8)

[PSC12] Vallabh Sambamurthy Pankaj Setia, Balaji Rajagopalan and Roger Calantone. How Peripheral Developers Contribute to Open-Source Software Development. *Information Systems Research*, 23:pp. 144–163, 2012. (cited on Page 2)

[QHB11] S.C. Cheung Qiaona Hong, Sunghun Kim and Christian Bird. Understanding a developer social network and its evolution. *IEEE International Conference on Software Maintenance (ICSM)*, 27:pp. 323 –332, 2011. (cited on Page 1)

[SLT09] Federico Barrero Sergio L. Toral, Rocío Martínez Torres. Modelling Mailing List Behaviour in Open Source Projects: the Case of ARM Embedded Linux. *Journal of Universal Computer Science*, 15(3), 2009. (cited on Page 6)

[TOS10] Filip Agneessens Tore Opsahl and John Skvoretz. Node centrality in weighted networks: Generalizing degree and shortest paths. *Social Networks*, 32(3):pp 245–251, 2010. (cited on Page 12)

[TR] Eric W. Weisstein Todd Rowland. Mathworld–a wolfram web resource: ”tensor”. http://mathworld.wolfram.com/Tensor.html. Accessed: 2019-08-15. (cited on Page 16)

[Weia] Eric W. Weisstein. Mathworld–a wolfram web resource: ”graph”. http://mathworld.wolfram.com/Graph.html. Accessed: 2019-08-18. (cited on Page 9)

[Weib] Eric W. Weisstein. Mathworld–a wolfram web resource: ”perron-frobenius theorem”. http://mathworld.wolfram.com/Perron-FrobeniusTheorem.html. Accessed: 2019-08-13. (cited on Page 13)

[Weic] Eric W. Weisstein. Mathworld–a wolfram web resource: ”strongly connected digraph”. http://mathworld.wolfram.com/StronglyConnectedDigraph.html. Accessed: 2019-08-15. (cited on Page 14)

[Weid] Eric W. Weisstein. Mathworld–a wolfram web resource: ”undirected graph”. http://mathworld.wolfram.com/UndirectedGraph.html. Accessed: 2019-08-18. (cited on Page 11)

[Weie] Eric W. Weisstein. Mathworld–a wolfram web resource: ”weighted graph”. http://mathworld.wolfram.com/WeightedGraph.html. Accessed: 2019-08-18. (cited on Page 11)

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 Bachelorarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Anselm Fehnker

Münster, 21. September 2019