HTBLA Wien 16

Höhere Lehranstalt für Informationstechnologie

Ausbildungsschwerpunkt Netzwerktechnik

DIPLOMARBEIT

Gesamtprojekt

Paenko

Backend Programmierung *Kevin Per 5AHITN* Betreuer: Prof. DI. Robert Baumgartner, MBA

Frontend Programmierung *Florian Hanko 5AHITN* Betreuer: Prof. DI. Robert Baumgartner, MBA

Abgabevermerk:

Datum: 04.04.2017 übernommen von:

KURZFASSUNG

Die Paenko Datenbank ist eine ausfallsichere, dokumentenorientierte NoSQL Datenbank basierend auf einer Implementation des Raft-Konsens Algorithmus in der Programmiersprache Rust. Die Datenbank soll eine sichere Alternative zu bereits bestehende Datenbanken wie zum Beispiel MongoDb oder MariaDb sein. Diese beiden Datenbanksysteme sind in ihrer Funktion Datensätze zu replizieren in mancher Hinsicht limitiert. Wir wollen mit der Datenbank ein System implementieren, das unlimitiert ist und deswegen beliebig groß skaliert werden kann und das unabhängig vom geographischen Standort der Server.

Die Diplomarbeit umfasst vier Programme. Die Datenbank selbst, die ACID Kriterien erfüllt und den Konsens-Algorithmus verwendet, der von uns um zahlreiche Datenbank Features erweitert wurde. Eine C# Library, die Kommunikation mit der Datenbank aus einem C# Programm heraus vereinfacht, ein Management Programm, welches in C# mithilfe der Library implementiert wurde und ein Webinterface, das auf allen Datenbankservern läuft. Beide dieser Management-Tools sollen dabei helfen, ein laufendes Paenko Datenbanksystem zu konfigurieren. Das gesamte Projekt ist Open Source und auf GitHub verfügbar.

ABSTRACT

The Paenko database is a failsafe, document-oriented, NoSQL database based on an implementation of the consensus algorithm Raft in the programming language Rust. Our database is supposed to be an alternative to database systems like MongoDb and MariaDb. These two systems have several limits when it comes to replicating data on multiple servers. We do not want our database to have such limits because our goal is to implement a system that is limitless und fully scalable, regardless of the physical location of the used database servers.

Our diploma project consists of four programs: the database itself, which is based on the Raft algorithm that was expanded to allow some useful features and satisfy the ACID criteria, a C# library, which is a database driver and simplifies using the database in a C# application, a two management applications, one of which is a C# application that was created using our library and the other a web interface that is hosted on every database server. Both programs allow database configuration over an appealing graphical user interface. All of the programs in this project are Open Source.

EHRENWÖRTLICHE ERKLÄRUNG

Ich versichere,

- dass ich meinen Anteil an dieser Diplomarbeit selbstständig verfasst habe,
- dass ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe
- und mich auch sonst keiner unerlaubten Hilfe bzw. Hilfsmittel bedient habe.

Wien, am 04.04.2017

Kevin Per	Florian Hanko

INHALTSVERZEICHNIS

KURZFASSUNG 2

ABSTRACT 3

EHRENWÖRTLICHE ERKLÄRUNG 4

1 MOTIVATION 9

2 IDEE 10

3 ZIELSETZUNG 12

3.1 HAUPTZIELE 12

4 ANWENDUNGSSZENARIEN 13

4.1 ANWENDUNG IM GESUNDHEITSBEREICH 13

4.2 ANWENDUNG IN DER LOGISTIK 13

5 KONKURRENZANALYSE 14

5.1 APOLLO VON FACEBOOK 14

5.2 SPANNER VON GOOGLE 14

6 DAS PAENKO ECOSYSTEM 15

7 KONSENSALGORITHMUS RAFT 16

7.1 STATEMACHINE UND LOG 16

7.2 ROLLENVERTEILUNG 17

7.3 FEHLERTOLERANZ DURCH TIMEOUTS 17

7.4 KOMMUNIKATION 18

7.5 LOG KONSISTENZ 18

8 IMPLEMENTATION VON RAFT-RS 20

8.1 STATEMACHINE 21

8.2 STATEMACHINE - APPLY() UND QUERY() 22

8.3 Log 23

9 TRANSAKTIONEN 24

9.1 Atomicity 24

9.2 Consistency 24

9.3 Isolation 24

9.4 Durability 25

9.5 Kommunikation 25

9.6 Implementation 26

10 DYNAMIC PEER ADDING 28

11 MULTI-LOG 30

11.1 Implementation 30

12 TRANSPARENZ 33

13 KOMMUNIKATION UND REST-SCHNITTSTELLE 35

13.1 Probleme 35

13.2 Lösung 35

13.3 Implementation 36

13.4 REST-API 37

14 MÖGLICHE ERWEITERUNGEN 39

14.1 Query Language 39

14.2 Multi User Konfiguration 39

14.3 Client - Prioritäten 39

14.4 SSL - Http Clients 39

14.5 SSL - Peers 39

14.6 Crosslog - Transactions 39

14.7 REMOVESERVER 40

15 TESTING 41

16 SICHERHEITSKONZEPTE 42

16.1 IMPLEMENTATION 42

17 KONFIGURATION 44

18 CLI-REFERENZ 46

19 USER-HANDBUCH 47

19.1 GETTING STARTED 47

19.2 DOCKER 48

20 C# LIBRARY 49

20.1 CRUD OPERATIONEN 49

20.2 GEOTRACKING 49

20.3 HEALTHCHECKS 50

20.4 ANWENDUNG 50

20.5 NODE 52

20.5.1 Document 53

20.5.2 UuidManager 53

20.5.3 Benutzungsbeispiel 54

21 C# MANAGER 55

21.1 ENTWICKLUNG 56

22 C# WEBINTERFACE 58

22.1 FUNKTION 58

23 MÖGLICHE FRONTEND-ERWEITERUNGEN 60

23.1 DATENBANK-TREIBER 60

24 WETTBEWERBE UND FÖRDERUNGEN 61

24.1 ITS AWARD 61

24.2 AXAWARD 61

24.3 JUGEND INNOVATIV 61

24.4 NETIDEE FÖRDERUNG 62

25 SOZIALE NETZWERKE 63

25.1 FACEBOOK 63

25.2 TRELLO 63

25.3 GITHUB 63

26 LESSONS LEARNED 65

VERZEICHNIS DER TABELLEN 66

VERZEICHNIS DER ABBILDUNGEN 67

ANHANG 68

ARBEITSAUFTEILUNG 68

ARBEITSPROTOKOLL – KEVIN PER 71

ARBEITSPROTOKOLL – FLORIAN HANKO 73

DIPLOMARBEITSANTRAG 75

1 MOTIVATION

Wir leben heutzutage in einem digitalen Universum. Informationen und Daten spielen eine bedeutende Rolle in Bezug auf die Qualität unseres Lebens. Gleichzeitig erleben wir eine explosionsartige Vermehrung von Daten, die übermittelt, gespeichert und ausgewertet werden müssen. Diese Daten sind weitgehend unstrukturiert. Sie fallen durch Social Media, Mobile Geräte, dem Internet of Things, dem World Wide Web, Business-to-Business Transaktionen, Digitalisierung und vieles mehr an.

Es wird geschätzt, dass bereits 2015 mehrere Zettabytes an Daten angefallen sind. Das ist eine Zahl mit 21 Nullen. Man vermutet, dass 2020 etwa 40 Zettabytes erreicht werden[1]. Diese Daten sind, wenn überhaupt, nur semi-strukturiert. Daher ist ein schemaorientierter Ansatz, wie er zum Beispiel bei relationalen Datenbanken verwendet wird, zur Speicherung und Abfrage nicht praktikabel.

Aufgrund dieser Situation haben wie uns einige Fragen gestellt: Wie sollen diese Daten gespeichert werden? Wie können sie ausgewertet werden? Wie kann Datenverlust verhindert werden, d. h. wie können diese Daten ausfallsicher verfügbar gemacht werden?

Mit unserem Projekt wollten wir diesen Fragen nachgehen und eine mögliche Lösung entwerfen und diese auch als Proof of Concept implementieren.

[1] https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm

2 IDEE

Bevor wir mit der Implementation begonnen haben, haben wir im Internet sehr viel über verteilte Datenbanken geforscht. Dabei haben wir uns aufgrund der obigen Überlegungen auf NoSQL Datenbanken konzentriert.

Wir fanden heraus, dass es viele verschiedene NoSQL Implementierungen gibt. Diese können nach ihrem Datenmodell in diverse Kategorien eingeteilt werden. Dabei stellen dokumentenorientierte Datenbanken wohl die bekannteste Kategorie dar. Ein Vorteil dieser Art von Datenspeicherung ist die große Flexibilität. Sowohl Key-Value Paare als auch semistrukturierte Daten mit JSON, XML und anderen Formaten sowie reine Textdateien können abgespeichert werden. Jede Speicherung kann anders erfolgen.

Daher war rasch klar, dass PaenkoDB eine dokumentenorientierte NoSQL Datenbank werden wird.

Ein wesentlich größeres Problem war die Art der Datenverteilung. Diese sollte sowohl über große geografische Distanzen erfolgen können als auch jederzeit die Konsistenz der Daten sicherstellen.

Es gibt zwei grundlegende Arten die Daten zu verteilen:

Sharding: Die Daten werden auf die Server derart verteilt, dass jeder Server für eine Teilmenge der Daten zuständig ist.

Replikation: Die Daten werden vollständig auf die Knotenrechner kopiert. Bei diesem Ansatz werden des Weiteren noch Master-Slave Systeme und Peer-to-Peer Systeme unterschieden.

Wir fanden, dass diese beiden Ansätze zu kurz greifen, da beim Sharding und bei der Replikation die Ausfallssicherheit nicht sehr hoch ist. So können zum Beispiel Sharding Rechner ausfallen. Damit stehen diese Daten dann nicht zur Verfügung. Bei der Replikation kann der Master ausfallen. Lediglich Peer-to-Peer Systeme bieten eine optimale Sicherheit, da sie beliebig erweiterbar sind und damit skalieren.

Auf der Suche nach einem geeigneten Algorithmus stießen wir auf den Raft Algorithmus[1], der 2014 von Diego Ongaro und John Ousterhout an der Stanford Universität erfunden wurde. Dieser Algorithmus erlaubt es Aufgaben zwischen mehreren PCs zu synchronisieren. Wir entschieden uns, diesen Algorithmus zur Basis unserer Datenbank zu machen. Wichtige Erweiterungen wie Transaktionen und Multilogs würden wir selbst hinzufügen müssen.

Nachdem die prinzipielle Machbarkeit geklärt war, betrieben wir Marktforschung und fanden heraus, dass es derzeit wohl keine Produkte mit der Funktionalität von PaenkoDB gibt.

Basierend auf den Ergebnissen unserer Recherche haben wir uns folgende Ziele für PaenkoDB festgelegt die im Kapitel Zielsetzung beschrieben werden.

[1] https://ramcloud.stanford.edu/~ongaro/thesis.pdf

3 ZIELSETZUNG

3.1 Hauptziele

3.1.1.1 Entwickeln der verteilten Datenbank

Es soll eine massiv verteilte NoSQL Datenbank entwickelt werden, die Daten als Dokumente abspeichert. Alle Datensätze sollen von allen Datenbank-Nodes parallel gespeichert und synchronisiert werden.

3.1.1.2 Kommunikation zwischen Datenbank-Nodes

Es soll ein Protokoll für die Kommunikation zwischen den Datenbank-Nodes zur Verfügung gestellt werden.

3.1.1.3 Entwickeln des User Interfaces

Damit auf Daten der Datenbank zugegriffen werden kann, werden diese über ein API auf Basis der REST-Architektur zur Verfügung gestellt. Mithilfe dieses APIs können programmiersprachenunabhängig Applikationen erstellt werden, die mit den Datenbank-Nodes kommunizieren.

Wir stellen eine C# Library zur Verfügung, welche Funktionen für die Kommunikation mit dem API implementiert und somit Manipulationen der Datenbank ermöglicht.

3.1.1.4 C# Verwaltungssoftware

Wir stellen zwei Applikationen zur Verwaltung unserer Datenbank zur Verfügung. Eine C# Applikation in welcher Datenbank-Nodes geographisch auf eine Google-Maps Karte visualisiert werden und ein Webinterface, das man auch auf Systemen nutzen kann auf denen die C# Applikation nicht läuft.

3.1.1.5 Entwickeln einer User Authentifizierung

Um für Sicherheit der gespeicherten Daten zu sorgen, besitzt unsere Datenbank eine User-Authentifizierung. Der User muss sich also mit Benutzername und Passwort anmelden um Zugriff auf seine Daten zu erlangen.

4 ANWENDUNGSSZENARIEN

PaenkoDB ist sehr flexibel und somit in vielen Bereichen anwendbar sein, doch besonders gut eignet sich PaenkoDB in Bereichen in denen es um Sicherheit und Erreichbarkeit der Daten geht.

4.1 Anwendung im Gesundheitsbereich

PaenkoDB ist perfekt für die Anwendung in Krankenhäusern. Zum Beispiel für das Organisieren von Patientendaten. Jeder Patient kann eine Id bekommen, die für seine Daten auf der Datenbank steht. Mit dieser Id kann er nun in jedes Krankenhaus gehen, das einen Paenko Node im selben System besitzt und dort kann ein Mitarbeiter mit Hilfe seiner Id die Patientendaten abrufen die sich am lokalen Node befinden. Also auch falls es einen Netzwerkfehler geben sollte befinden sich alle Patientendaten immer lokal und können vom Node abgerufen werden.

4.2 Anwendung in der Logistik

Ein einheitlich konsistentes System ist in der Logistik besonders wichtig. Da PaenkoDb, durch ständige Replikation, genauso ein System zur Verfügung stellt kann die Datenbank in diesem Bereich eingesetzt werden. Konkurrenzprodukte in diesem Bereich werben mit präzisen und zuverlässigen Informationsabfragen, welche aber nur zentralisiert ablaufen, PaenkoDb hat also den Vorteil, dass Informationen an mehreren Standorten repliziert sind.

5 KONKURRENZANALYSE

Es werden derzeit zwei Produkte entwickelt, die PaenkoDB ähnlich sind:

5.1 Apollo von Facebook

Diese Datenbank verwendet ebenfalls den Raft Algorithmus, aber nur zur Abstimmung. Apollo repliziert Daten nicht auf jeden Rechner so wie unsere Datenbank sondern verwendet Sharding. Beim Sharding werden Teilmengen von Daten auf Server aufgeteilt. Damit hat man größere Kapazitäten und auch mehr Rechenleistung.

5.2 Spanner von Google

Spanner ist nicht NoSQL sondern NewSQL und verwendet nicht wie PaenkoDB den Raft Algorithmus sondern Paxos. Paxos ist ein etwas älterer Konsens Algorithmus der um einiges komplizierter ist als Raft. Die Datenbank ist in Zonen aufgeteilt die sich über diesen Konsens Algorithmus austauschen.

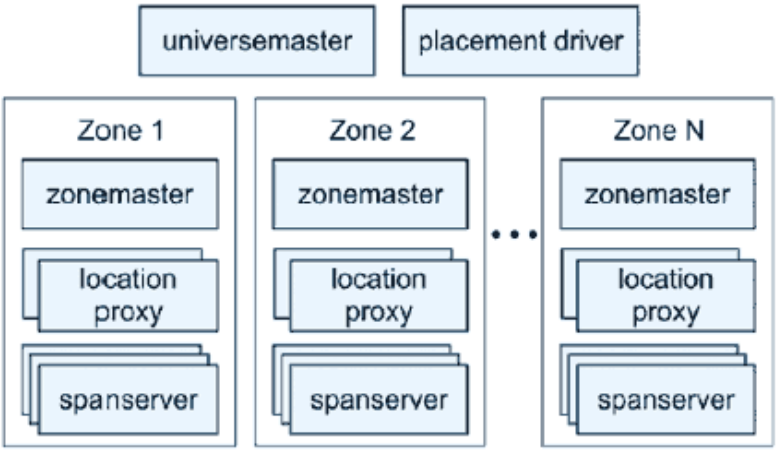


Abbildung 1: Google Spanner Architektur

6 Das Paenko Ecosystem

Unser Projekt besteht aus mehreren Teilen. Wir versuchen mit diesem Ecosystem eine für den Anwender und Entwickler praktikable Möglichkeit für den Umgang mit der Datenbank zu bieten. Die Hauptkomponenten dieses Systems sind das Datenbanksystem und die C# Library. Das Datenbanksystem ist verantwortlich für die Organisation der Daten und kümmert sich auch um die Zugriffe. Das Datenbanksystem bezeichnen wir als PaenkoDb.

Das Ökosystem besteht aber nicht nur aus einem Datenbanksystem, sondern auch aus einem Frontend. Hauptsächlich haben wir uns dabei auf die C#-Library fokussiert. Diese kann man in sein C#-Projekt einbinden und dient als Schnittstelle für die Kommunikation zu PaenkoDb. Aufbauend auf dieser Library haben wir eine Manager-Applikation entwickelt. Neben der C# Library, gibt es aber noch eine Weboberfläche.

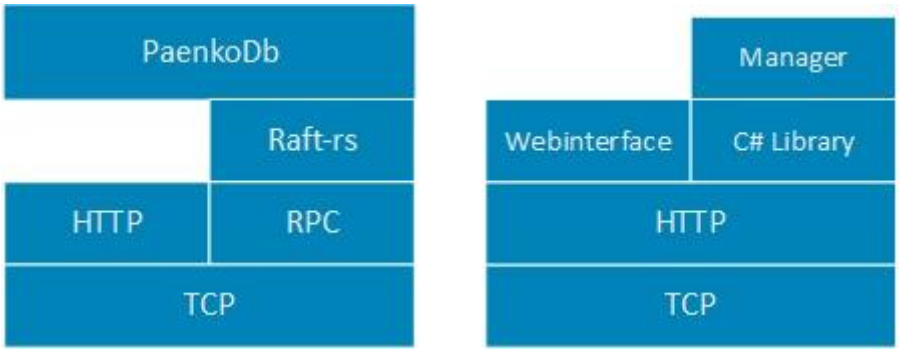


Abbildung 2: Architektur Diagramm von PaenkoDb(links) und Frontend(rechts)

7 KONSENSALGORITHMUS RAFT

Ein fehlertolerantes System besteht aus mehreren Teilnehmern, die unabhängig davon, ob Teilnehmer ausfallen, trotzdem funktioniert. Ein wichtiger Bestandteil eines fehlertoleranten Systems ist der Konsensalgorithmus.

Ein Konsensalgorithmus ist ein Algorithmus, der vorschreibt, wie die Kommunikation der Teilnehmer abläuft und wie mit Teilnehmer-Ausfällen umgegangen werden soll. Aus diesem Grund ist er fundamental für fehlertolerante Systeme.

Raft ist ein Algorithmus, der eine Alternative zu Paxos sein soll. Paxos ist ein etwas älterer Konsensalgorithmus, der bis zur Veröffentlichung von Raft der Standard war (Google-Spanner verwendet Paxos). Das Ziel eines Konsens ist es, dass sich mehrere Teilnehmer auf bestimmte Werte einigen. Ein Konsens ist besonders wichtig in fehlertoleranten verteilten Systemen. Einen solchen Konsens in der Praxis zu erreichen, ist jedoch ein schwieriges Unterfangen. Denn in der Realität fallen Knotenrechner gelegentlich aus, Antworten verzögern sich und ausgefallene Knotenrechner wollen wieder teilnehmen. Solche Umstände machen es problematisch auf einen Konsens zu kommen.

7.1 Statemachine und Log

Die Statemachine von PaenkoDb stellt sicher, dass die Zustände der Knotenrechner auf allen Servern konsistent sind. Jeder Teilnehmer hat eine Statemachine und ein Log. Ein Log besteht aus mehreren Entries. Jedes Entry enthält ein Kommando, das von der Statemachine ausgeführt wird. Angenommen die Aufgabe unserer Statemachine ist es Variablen zu setzen und unser erster Entry ist $x = 5$. Sobald dieses Entry auf allen Teilnehmern repliziert ist, hat jeder Knoten eine Variable mit $x = 5$. Würden wir noch einen Entry hinzufügen $x = 7$ und den Konsensalgorithmus diesen wieder replizieren lassen, hat jeder Teilnehmer das alte x überschrieben und der neue Wert wäre $x = 7$. Besonders wichtig ist, dass diese Entries auf allen Teilnehmern in der gleichen Reihenfolge sind, sonst könnte es passieren, dass der ursprünglich erste Entry zum Schluss ausgeführt wird und damit $x = 5$ ist, was nicht korrekt wäre. Der Konsensalgorithmus stellt also sicher,

dass alle Einträge auf allen Knoten vorhanden sind und die Reihenfolge der Entries im Log bei allen Teilnehmern gleich ist.

7.2 Rollenverteilung

Raft teilt seinen Teilnehmern Mitgliedsarten zu. Der Algorithmus unterscheidet zwischen Leader, Candidate und Follower (siehe Abbildung 3). Ein Leader besitzt immer das aktuellste Log. Follower können selbständig keine Entries zum Log hinzufügen. Alle Änderungen am Log müssen beim Leader erfolgen. Jedoch ein Leader zu sein ist kein persistentes Privileg, sondern sie werden gewählt. Falls es zu einem Leader Ausfall kommen sollte, wird ein anderer Teilnehmer diese Rolle übernehmen. Alle Anwärter für die Rolle des Leader sind im Candidate State.

Es darf aber nur einen Leader geben!

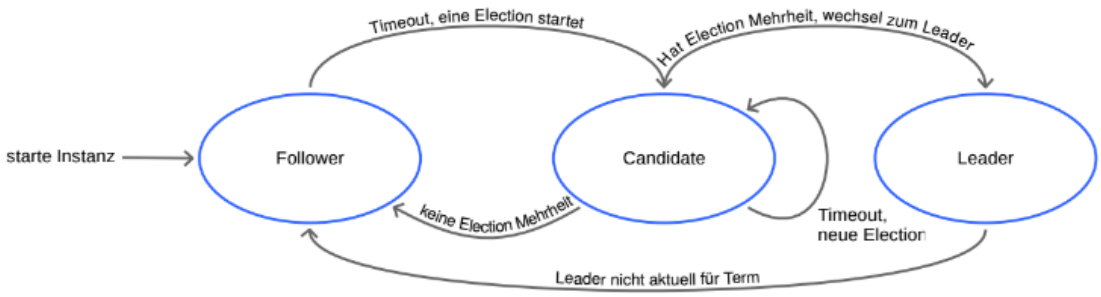


Abbildung 3: Ablauf der States in Raft

7.3 Fehlertoleranz durch Timeouts

Es kann passieren, dass ein Leader ausfällt. In der Zwischenzeit würde ein anderer Teilnehmer zum Leader werden. Würde der ursprüngliche Leader zurückkehren, gäbe es jetzt zwei Leader. Um die Bedingung, dass es unter allen Teilnehmer nur einen Leader geben darf, sicherzustellen, ist die Rolle eines Leaders zeitlich begrenzt auf eine *Term*. Terms gelten als Zeitspanne in Raft und sind numerisch, um sie zwischen Teilnehmern vergleichen zu können. Wenn ein Leader gewählt wurde, dann gilt das nur für eine Term. Es wird nur der Leader mit der höchsten Term beachtet, Leader mit älteren Terms werden automatisch zu Followern. Wenn also ein Leader einer älteren Term wieder am Konsens, bedingt durch

beispielsweise einen Ausfall, teilnimmt, erkennt er, dass ein anderer Node zum Leader geworden ist.

Um einen neuen Leader zu wählen, wartet jeder Teilnehmer sein *Election-Timeout* ab. Diese Zeitdauer ist zufällig und daher auf jedem Knoten unterschiedlich. Sobald dieses Timeout abgelaufen ist, geht der Follower in den Candidate State und erhöht seine Term um 1. Im Candidate State sendet er jedem Teilnehmer eine Anfrage um Leader zu werden. Sobald eine Mehrheit der Teilnehmer zugestimmt hat, konvertiert er zum Leader.

Ein Leader hat kein Election-Timeout, da er nicht zum Leader werden kann, weil er bereits ein Leader ist. Stattdessen hat er ein *Heartbeat-Timeout*. Anders als das zufällige Election-Timeout ist das Heartbeat-Timeout regelmäßig. Außerdem sollte das Election-Timeout von der Dauer her ein Vielfaches höher sein als das Heartbeat-Timeout. Der Heartbeat dient dazu, um den Follower mitzuteilen, dass der aktuelle Leader noch “am Leben ist” und die aktuelle Term noch aktuell ist. Sobald ein Follower ein Heartbeat empfangen hat, wird das Election-Timeout neugesetzt.

7.4 Kommunikation

Raft Server kommunizieren über Remote Procedure Calls. Raft sieht zwei RPC Typen vor. Ein *RequestVote RPC* für LeaderElections und *AppendEntries RPC* für das Replizieren von Log Entries und Heartbeats. Jedoch wurden weitere in der Raft-rs eingeführt. Alle implementierten RPC sind in *src/messages.capnp* definiert.

7.5 Log Konsistenz

Wie bereits erwähnt, besteht jedes Log aus mehreren Entries. Jedes Entry hat eine Position im Log, diese nennt man LogIndex. Vergleicht man zwei Entries, welche den gleichen LogIndex und Term haben, dann speichern sie das gleiche Kommando auf allen Knoten. Gleichzeitig sind alle Entries davor identisch. Diese Erkenntnis ist besonders relevant, wenn ein neuer Leader gewählt wurde und nicht alle Entries des Vorgängers repliziert wurden oder Follower ausgefallen sind. Denn bevor ein Leader neue Entries replizieren kann, müssen die Logs der Follower

“repariert” werden. Der Leader besitzt einen Counter für jeden Follower, der bestimmt, welche Entries vom Log des Leaders an den Follower im nächsten AppendEntries RPC gesendet werden. Das “AppendEntries RPC” ist ein Nachrichtenformat, damit der Leader neue Entries in das Log der Follower einfügen kann und eventuell die Logs der Follower auch “reinigen” kann.

Wenn ein Leader neu gewählt wurde, sind alle seine Counter für die Follower genauso lang wie sein Log. Vor jedem AppendEntries RPC erfolgt dann ein Konsistenz-Check bei jedem Follower. Wenn die Reaktion des Follower auf das AppendEntries des Leaders negativ ist, dekrementiert der Leader den Counter des entsprechenden Followers. Hinzu kommt wenn der Konsistenz Check beim Follower fehlschlägt, wird der aktuellste Entry gelöscht. Dieser Prozess wird solange wiederholt, bis der Follower ein aktuelles Log besitzt.

8 IMPLEMENTATION VON RAFT-RS

Raft-rs ist die Raft Implementation der Doktorarbeit von Diego Ongaro und John Ousterhout. Diese Implementation wurde von Andrew Hobden entwickelt und ist ein wichtiger Bestandteil der Diplomarbeit.

Der Raft-Cluster besteht aus 3 Nodes. Um das Bild zu vereinfachen, wird angenommen, dass Node 1 und Node 3 bereits miteinander verbunden sind. Damit sich Node 1 und Node 2 verbinden, senden sie sich gegenseitig ein *ServerConnectionPreamble*. Node 2 wird zum Leader gewählt und alle anderen Nodes werden zum Follower. Nachdem Node 2 erfolgreich zum Leader wurde, stürzt dieser in diesem Beispiel ab. Da Node 2 jetzt nicht mehr verfügbar ist und keine regelmäßige *Heartbeats* sendet, läuft das *ElectionTimeout* von Node 1 aus und er wechselt vom *Follower State* in den *Candidate State*. Node 1 sendet dann all seinen Peers eine *RequestVote*, um sich zum neuen Leader ernennen zu können. Sobald Node 1 von einer Mehrheit von Peers eine *RequestVoteSuccess* Antwort bekommen hat, darf er in den Leader State wechseln. Daraufhin beginnt Node 1 *AppendEntries Messages* zu senden und stellt sicher, dass das Log der Follower mit dem des Leaders gleich ist, indem er falsche Einträge entfernt und neue hinzufügt. Gibt es am Leader keine neuen Einträge, wird diese Nachricht trotzdem geschickt, denn diese dient in der Implementation als Heartbeat und setzt die *ElectionTimeouts* der Follower zurück. In unserer Simulation kommt dann der ursprüngliche Leader, Node 2, wieder zurück und möchte wieder dem Cluster beitreten. Er sendet wieder an all seine Peers ein *ServerConnectionPreamble*, um sich neu zu verbinden. Nachdem Node 2 wieder verfügbar ist, senden beide Nodes sich wieder ein *ServerConnectionPreamble*. Damit ist die ursprüngliche Connection wiederhergestellt und Node 2 erkennt durch die niedrigere Term, dass ein neuer Leader existiert und wechselt in den *Follower State*.

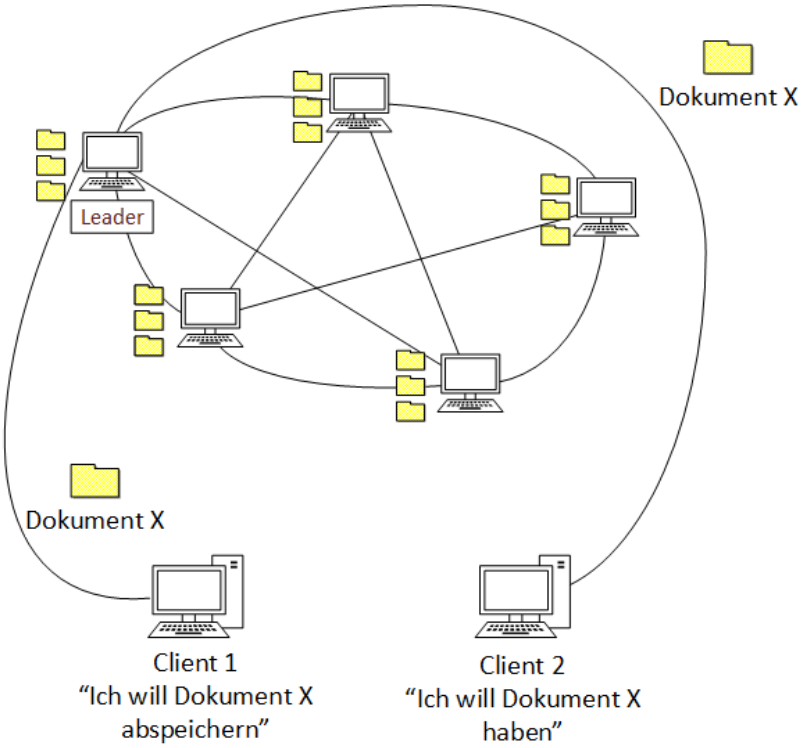


Abbildung 4: Überblick von PaenkoDb

8.1 Statemachine

Die Statemachine bzw. die DocumentStateMachine basiert auf einem Statemachine Interface, das auch StateMachine-Trait genannt wird. Das Statemachine Trait setzt einige Methoden voraus, um von der Raft Implementation verwendet werden zu können. Die wichtigsten zwei Methoden sind *apply* und *query*.

```
#[derive(Debug,Clone)]
pub struct DocumentStateMachine {
    log: Vec<DocumentRecord>,
    map: HashMap<DocumentId, Document>,
    volume: String,
    transaction_offset: usize,
}
```

Die *DocumentStatemachine* besteht folglich aus insgesamt vier Feldern. Das erste Feld *log* ist ein Vektor aus *DocumentRecords*. Ein DocumentRecord ist eine Aufzeichnung der Operationen, die zwischen den Nodes gesendet werden. Dieser ist gleichzusetzen mit einem LogEntry. Das zweite Feld ist *map*. Map ist eine HashMap mit den derzeit verfügbaren Dokumenten. Wenn ein Dokument also

gelöscht wird, dann wird ein DocumentRecord mit DELETE-Operation ins Log eingefügt und das jeweilige Dokument wird aus Map gelöscht. Das Feld *volume* ist ein String und gibt an in welchen Ordner die Dokumente bzw. den aktuellen Zustand der Statemachine gespeichert werden soll. Das letzte Feld von der DocumentStateMachine ist *transaction_offset* und dieses wird beim Transaktions-Rollback benötigt.

8.2 Statemachine - apply() und query()

Raft-rs interagiert mit der Statemachine über die Methoden *apply* und *query*. Diese zwei Methoden werden von Raft-rs aufgerufen, wenn der Server eine Client Nachricht empfängt, welches eine Anfrage oder Abfrage ist.

Dabei wird das Kommando an die Methode übergeben. Dieses dekodiert dann diese Bytes in ein Enum mit vier möglichen Zuständen(Get, Post, Remove und Put). Diese entsprechen der Operation, die man durchführen möchte und leitet die notwendigen Daten weiter, die in der Message gekapselt sind. Die apply-Methode ist hauptsächlich für die Kommandos, die zur Gruppe Anfragen gehören, zuständig. Das sind Post, Remove und Put. Wobei die query-Methode sich nur um die GET-Operation kümmert. Jedoch kann *apply* eine Abfrage auch an *query* weiterleiten.

```
fn apply(&mut self, new_value: &[u8]) -> Vec<u8> {
        let message = decode(&new_value).unwrap();

        let response = match message {
            Message::Get(_) => self.query(new_value),
            Message::Post(document) => self.post(document),
            Message::Remove(id) => self.remove(id),
            Message::Put(id, new_payload) => self.put(id, new_payload),
        };

        self.snapshot();

        response
    }
```

In den aus der apply-Methode aufgerufenen Methoden *self.post*, *self.remove* und *self.put* werden Aktionen durchgeführt, die den Zustand verändern. Zuerst wird in diesen Methoden ein Eintrag in das Log von der Statemachine hinzugefügt.

Dieser Eintrag ist ebenfalls im Log vom Knoten. Notwendig wird der Eintrag im Log der Statemachine erst bei Transaktionen bzw. sobald Transaktionen fehlschlagen, damit diese Einträge auch wieder „zurückgerollt“ werden können. Des Weiteren kümmern sich die Methoden um das *map*. Bei einer Post Message, wird also in map ein neues Dokument eingefügt. Bei einer Delete Message, wird aus der map ein Dokument gelöscht und bei einer Put Message, ein bestehendes verändert. Das vereinfacht, das Abfragen von Dokumenten.

8.3 Log

Das Log ist, neben der Statemachine, unentbehrlich für Raft. Während die Statemachine sich mit den Kommandos, Create, Read, Update und Delete von Dokumenten auseinandersetzt, beschäftigt sich das DocLog mit der Speicherung von Daten. Zusätzlich ist das Log eine Schnittstelle für den Konsensalgorithmus um neue Einträge hinzufügen und wieder zu löschen.

```
#[derive(Clone, Debug)]
pub struct DocLog {
    entries: Vec<(Term, Vec<u8>)>,
    logid: LogId,
    prefix: String,
}
```

Zwar werden Log-Entries auch in das Log der Statemachine hinzugefügt, diese sind aber irrelevant für Raft. Die notwendigen Informationen für den Algorithmus bzw. die Implementation des Algorithmus liefert das Log. Alle Werte des Logs müssen persistent sein, damit falls ein Knoten ausfällt und eventuell wieder in den Cluster aufgenommen wird, keine doppelten Informationen gesendet werden.

9 TRANSAKTIONEN

Eine Transaktion ist eine Gruppe von Nachrichten, die entweder alle erfolgreich sind oder gar nicht ausgeführt werden. Der große Vorteil ist, dass es zu keinem inkonsistenten Zustand kommt. Denn es wird von einem konsistenten Zustand in den anderen gewechselt.

Transaktionen unterliegen dem ACID Prinzip. ACID steht für Atomicity, Consistency, Isolation und Durability.

9.1 Atomicity

Eine Transaktion wird in PaenkoDb gestartet durch eine *TransactionBegin*- und beendet durch eine *TransactionCommit*- oder *TransactionRollback*-Nachricht. Dabei muss unterschieden werden in Client- und Peer-Nachrichten. Ein *Client-TransactionBegin* initiiert eine Transaktion, aber die Peer Nachrichten verteilen diese Nachricht an alle Knoten. Wenn ein *TransactionBegin* erfolgreich iniitiert wurde, sendet der Node eine *TransactionId* an den Client zurück. Diese Id kennzeichnet Nachrichten, die zur Transaktionen gehören und muss bei jeder Client-Nachricht angegeben werden, um die Nachrichten-Gruppe zu identifizieren.

9.2 Consistency

TransactionCommit und *TransactionRollback* beendet die Transaktion. Der Client kann mit diesen Nachrichten dem Cluster mitteilen, ob die Transaktion erfolgreich ist oder zurückgerollt werden soll.

9.3 Isolation

PaenkoDb unterstützt nur eine Transaktion pro Log. Nachrichten, die nicht die *TransactionId* enthalten oder nicht die der Transaktion, werden in eine Warteschlange gelegt und die Verbindung ist blockiert. Sobald die Transaktion abgeschlossen ist, wird die Warteschlange abgearbeitet und die Clients, die nicht an der Transaktion beteiligt waren, bekommen zeitverzögert ihre Antworten.

9.4 Durability

Abgeschlossene Transaktionen sind unmittelbar im Log und damit dauerhaft gespeichert. Außerdem ist es durch die Zeitverzögerung sogar wahrscheinlich, dass Transaktionen-Nachrichten bereits an die Followern übertragen wurden und somit sobald die Beendigung der Transaktion erfolgt, sofort auf allen Knoten verfügbar sind. Wenn der Leader während einer Transaktion ausfällt, rollt der nächste Leader alle Änderungen zurück.

9.5 Kommunikation

Der Client sendet eine Nachricht, um die Transaktion zu starten. Sobald der Leader diese Nachricht empfangen hat, bereitet er die Transaktion vor und sendet an alle Follower eine Message, dass sie es ihm gleich tun sollen. Diese Transaktion gilt nur für ein Log und alle Messages an den Leader benötigen eine Session, um zwischen Transaction-Messages und gewöhnlichen Messages unterscheiden zu können. Wenn die Session nicht gleich mit der *TransactionSession* ist, dann wird die Message in eine Warteschlange abgelegt. Sobald der Leader eine Rollback- oder Commit-Nachricht an alle Follower gesendet hat, ist die Transaktion beendet und er beginnt damit die Message-Queue abzuarbeiten.

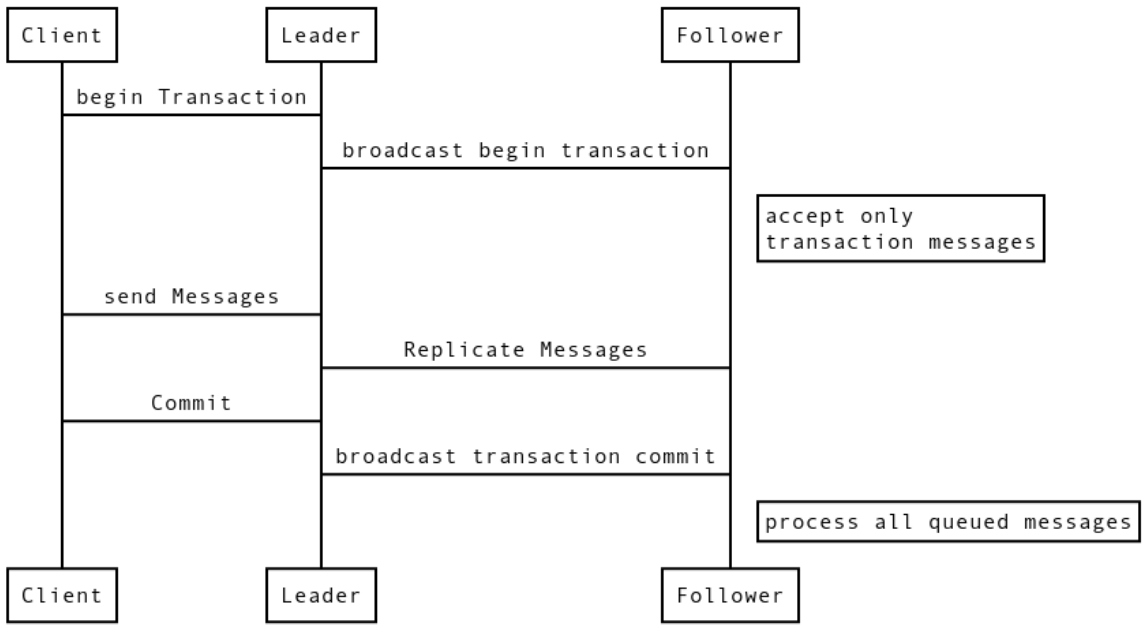


Abbildung 5: Ablauf einer Transaktion

Wie auch bei den anderen *Proposals* müssen die Messages *TransactionBegin*, *TransactionCommit* und *TransactionRollback* zum Leader gesendet werden. Falls die erwähnten Messages zu einem Follower gesendet werden, wird ein Fehler zurückgesendet. Der Client-Handler der Datenbank fängt diesen Fehler ab und extrahiert aus dem Fehler den Leader. Daraufhin wird die Message nochmal gesendet, jedoch diesmal an den Leader. Im Endeffekt, wenn eine Transaction Message zu einem Follower gesendet, wird diese an einem Leader weitergeleitet.

9.6 Implementation

Für die Verwaltung von Transaktionen ist der *TransactionManager* zuständig. Jede Consensus Instanz erstellt seinen eigenen. Dadurch kann pro Log eine Transaktion gestartet werden.

Der Manager implementiert die Methoden, um Transaktionen zu starten, beenden und abzubrechen. Außerdem speichert er Informationen über den letzten konsistenten Zustand der Datenbank, um im Falle eines *Rollbacks*, alle Änderungen zurücksetzen zu können.

```
#[derive(Clone)]
pub struct TransactionManager {
    /// Whether a transaction is running
    pub is_active: bool,
    /// The ID of the current transaction. If `None`, no transaction is runn
ing
    pub session: Option<TransactionId>,
    /// The amount of the messages which has been applied during transaction
    counter: usize,
    /// The commit_index before the transaction
    commit_index: LogIndex,
    /// The last_applied index before the transaction
    last_applied: LogIndex,
    /// The follower_state_min index before the transaction
    follower_state_min: Option<LogIndex>,
}
```

Die Nachrichten der zweiten Gruppe ermöglichen einem Client, mit Transaktionen umzugehen. Sobald die Consensus-Instanz Transaction Messages empfängt, werden die entsprechenden Methoden im *TransactionManager* aufgerufen. In der Implementation unterscheiden wir dabei in zwei verschieden Transaktion-

Nachrichten-Gruppen. Die erste Gruppe umfasst Messages für die Kommunikation zwischen Peers. Die zweite Gruppe ist für die Kommunikation der Clients.

Die Nachrichten der ersten Gruppe dienen dazu, um andere Nodes über Transaktionen zu informieren. Diese Nachrichten werden nur vom Leader an alle Follower gesendet.

Um Nachrichten zu einer Transaktion Kette zusammenführen, müssen die Nachrichten von Operationen wie Post, Update und Delete, die *TransactionId* der gerade aktiven Transaktion enthalten. Die Überprüfung, ob eine Transaktion aktiv ist und ob sie eventuell in die Warteschlange abgelegt werden, erfolgt dann beim Empfang eines *Proposals*, welches ein Wrapper für die verschiedenen Nachrichtentypen ist. Wenn keine Transaktion aktiv ist, wird die *TransactionId* ignoriert.

10 DYNAMIC PEER ADDING

In der ursprünglichen Version von Raft-rs müssen alle Peers vor dem Start der Applikation definiert sein. Um in PaenkoDb dynamisch, also zur Laufzeit, neue Peers hinzufügen zu können, muss in der Konfiguration ein neues Feld hinzugefügt werden, dass einen Node deklariert mit dem sich der neue Peer verbinden soll. Danach werden die Peers der Peers ausgetauscht, damit der neue Peer sich ebenfalls mit den restlichen aus dem Cluster verbinden kann.

config.toml

```
...
[dynamic_peer]
node_id= 1
node_address = "leader:9000"
...
```

Dazu wird bevor der Node sich über ein *ServerConnectionPreamble* verbindet, eine *AddServer-Message* gesendet, die Einstellungen für einen neuen Peer vornimmt. Der Node, welcher angesprochen wurde, reagiert dann, indem er dem neuen Node eine *ServerConnectionPreamble* sendet. Dabei werden auch dem neuen Peer alle Peers des Clusters gesendet, damit er sich auch mit denen verbinden kann.

```
connection_preamble::id::Which::ServerAdd(peer) => {
	let peer = try!(peer);
	let peer_id = ServerId(peer.get_id());
	let community_string = peer.get_community().unwrap();

	// Not the source address of this connection, but the
	// address the peer tells us it's listening on.
	let peer_addr = SocketAddr::from_str(try!(peer.get_addr())).unwrap();
	scoped_debug!("received new connection from {:?} ({})",
	peer_id,
	peer_addr);

if self.community_string == community_string {

	if !self.log_manager.check_peer_exists(peer_id) {
		self.log_manager.add_peer(peer_id, peer_addr);
		self.add_peer_static(event_loop, peer_id, peer_addr).unwrap();
	} else {
		// Was already connected
		scoped_debug!("Dynamic peer wants to reconnect {:?}",
		peer_addr);
	}
} else {
	scoped_warn!("The community_string is not equal and therefore \
	the connection will need to be established");
}
}
```

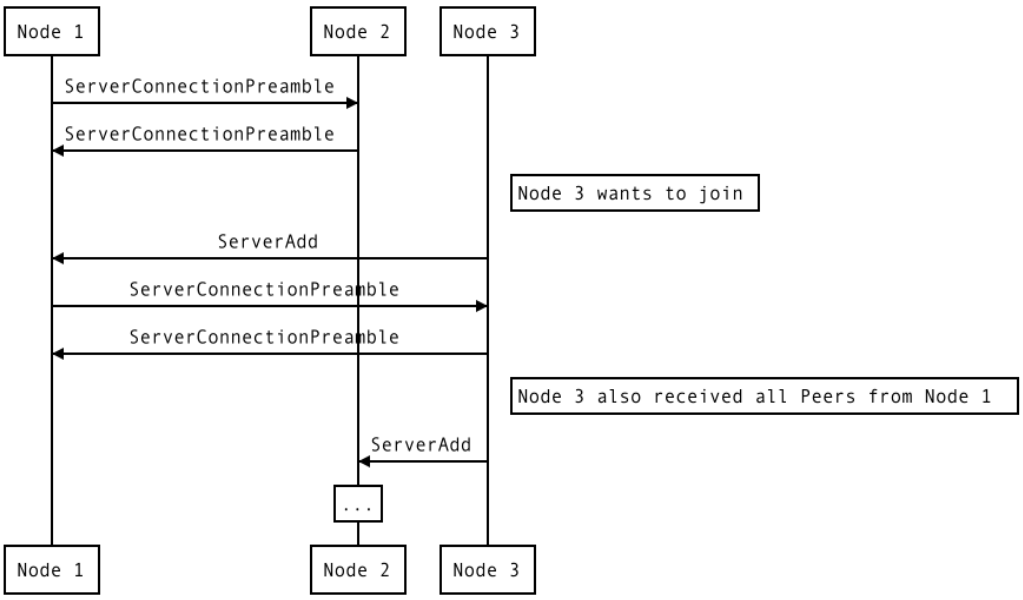


Abbildung 6: Ablauf von ServerAdd

11 MULTI-LOG

Die Implementation von Raft sieht den Einsatz von nur einem Log vor. Es kann jedoch sinnvoll sein, Dokumente in mehrere *Logs* zu teilen, da bei vielen Transaktionen, die Latenz drastisch erhöht werden kann.

In PaenkoDb haben wir deshalb Multilog implementiert. Multilog ermöglicht es mehrere *Logs* parallel im Betrieb zu haben. Das erlaubt dem Nutzer die logische Trennung von Dokumente und erinnert an das Konzept von relationalen Datenbanken.

Zur Identifikation von Logs, besitzt jede Instanz eine Id. Diese Id muss auf allen Nodes statisch konfiguriert werden und kann sich nicht ändern. Außerdem muss jedes Request eines Clients diese *LogId* enthalten.

11.1 Implementation

```
pub struct LogManager<L, M>
    where L: Log,
          M: StateMachine
{
    peers: Arc<RwLock<HashMap<ServerId, SocketAddr>>>,
    pub consensus: HashMap<LogId, Consensus<L, M>>,
}
```

Für die Verwaltung der verschiedenen Logs ist der *LogManager* zuständig. Dieser dient als Abstraktionsschicht. Das Herz des *LogManager* ist das *consensus* Feld, dass aus einer Hashmap aus Consensus-Instanzen besteht. Identifiziert werden diese Instanzen über die *LogId*. Außerdem verfügt der *LogManager* über zwei wichtige Methoden: *apply_client_message* und *apply_peer_message*. Diese werden vom Server-Instanz aufgerufen, wenn eine Client und Peer Nachricht empfangen wurde. In diesen Methoden vom *LogManager* wird dann die *LogId* aus der Message gelesen und dann die entsprechende Methode *apply_client_message* oder *apply_peer_message* von der Consensus-Instanz aufgerufen.

```
 pub fn apply_client_message<S>(&mut self,
                                from: ClientId,
                                message: &Reader<S>,
                                actions: &mut Actions)
        where S: ReaderSegments
    {
        let reader = message.get_root::<client_request::Reader>().unwrap();
        let log_id = LogId(Uuid::from_bytes(reader.get_log_id().unwrap()).un
wrap());

        scoped_trace!("Received client message on log {:?}", log_id);

        let mut cons = self.consensus.get_mut(&log_id).unwrap();

        cons.apply_client_message(from, &reader, actions);
    }

 pub fn apply_peer_message<S>(&mut self,
                              from: ServerId,
                              message: &Reader<S>,
                              actions: &mut Actions)
        where S: ReaderSegments
    {
        let reader = message.get_root::<message::Reader>().unwrap();

        let logid_in_bytes = reader.get_log_id()
            .expect("LogId property was not set in the message");
        let id = match Uuid::from_bytes(logid_in_bytes) {
            Ok(id) => id,
            Err(_) => {
                scoped_warn!("Received a message with an invalid LogId; Disc
arding it");
                return;
            }
        };

        let log_id = LogId(id);

        let mut cons = self.consensus
            .get_mut(&log_id)
            .expect(&format!("There is not consensus instance with this {:?}
", log_id));

        cons.apply_peer_message(from, &reader, actions);
    }
```

Neben der Implementation des Log-Managers mussten auch *Heartbeats*- und *Election-Timeouts* überarbeitet werden. Raft-rs benutzt für die Netzwerk Kommunikation, die Library *mio*. Ein Problem, das bei mio passieren könnte ist, wenn ein Log eine Vielzahl von Messages abarbeiten muss, können alle anderen Logs ausgebremst werden, da mio Single Threaded ist. Außerdem war es eine Schwierigkeit, dafür zu sorgen, dass Heart-Timeouts gleichmäßig timeouten. Der Grund für diese Schwierigkeit war ein eigenartiges Verhalten bei der Setzung von Timeouts in der Event-Loop von der Library Mio, bei der wir die Timeouts registrieren mussten. Während bei der ursprünglichen Version der Raft Implementation immer nur ein *Heartbeat-Timeout* gesetzt war, mussten wir für jedes Log am Leader ein *Heartbeat-Timeout* registrieren. Der erste Ansatz war einfach ein Tuple aus *LogId* und Timeout zu registrieren. Das führte aber zu dem erwähnten eigenartigen Verhalten. Wenn man ein Tuple in mio registriert, timen die restlichen *Heartbeat-Timeouts* nicht mehr aus. Das konnten wir lösen, indem wir statt ein Tuple zu übergeben, das *Heartbeat-Timeout* im *ServerTimeout-Enum* um *LogId* erweitert haben.

```
#[derive(Clone, Copy, Debug, Eq, PartialEq, Hash)]
pub enum ServerTimeout {
    Consensus(LogId, ConsensusTimeout),
    Reconnect(Token),
}

# Registrierung des Timeouts
let handle = event_loop.timeout_ms(ServerTimeout::Consensus(lid, timeout), d
uration).unwrap();
```

12 TRANSPARENZ

In der originalen Version von raft-rs musste dem Client der Leader bekannt sein, damit der Leader angesprochen werden kann. Die Methode *send_message* würde mit einer Schleife versuchen die Nachricht solange an alle dem Client bekannten Nodes zu senden, bis sie den Leader erreicht. Jedoch könnte es für den Client Anwender ein Problem darstellen an eine Liste alle Peers zu kommen. Vor allem wenn dynamisch neue Peers hinzukommen würden.

```
let mut members = self.cluster.iter().cloned();
loop {

            let mut connection = match self.leader_connection.take() {
                Some(cxn) => {
                   …
                }
                None => {
                   …
                }
            };
}
```

Um dieses Problem zu lösen, sollte sobald die erste Nachricht fehlschlägt, die darauf kommende zweite Nachricht, sprich der neue Versuch, gleich an den Leader gesendet werden. Denn ein Follower sollte den Leader im aktuellen Term kennen. Sobald also die erste Nachricht scheitert, weil ein Follower angesprochen wurde(*RaftError::ClusterViolation*), wird aus der Fehlermeldung der Leader extrahiert und die eigene Methode rekursiv aufgerufen. Das gilt für alle Client-Nachrichten, inklusive Transaktions-Nachrichten.

```
let payload = encode(&Message::Post(document.clone()), SizeLimit::Infinite).
unwrap();

let response = match client.propose(session, payload.as_slice()) {
            Ok(res) => res,
            Err(RError::Raft(RaftError::ClusterViolation(ref leader_str))) =
> {
                return Handler::post(&parse_addr(&leader_str),
                                     &username,
                                     &plain_password,
```

```
                                document,
                                session,
                                lid);
        }
        Err(err) => return Err(err),
    };
```

13 KOMMUNIKATION UND REST-SCHNITTSTELLE

Zur Kommunikation von Client und Peers wurde Capnproto verwendet. Capnproto ist eine Applikation um aus generischen Nachrichtenformaten, Programm-Code zu generieren um plattformunabhängige Nachrichten senden und empfangen zu können. Die Verwendung von Capnproto für andere Plattformen kann aber zu Problemen führen, wenn Programmiersprachen nicht die notwendigen Libraries besitzen, um Capnproto Nachrichten zu kompilieren. Aus diesem Grund, empfiehlt es sich eine plattformunabhängige Schnittstelle zu implementieren. In PaenkoDb ist deswegen eine HTTP REST-API Schnittstelle implementiert. Die Schnittstelle verfügt über eine API um sich einloggen, Messages abzusetzen, mit Transaktionen umgehen zu können und Meta-Information über Nodes abfragen zu können. Dazu wurde das Webframework Iron verwendet. Dieses ermöglicht uns Routen zu definieren und einfach auf den Body und URL von HTTP-Request zugreifen zu können.

13.1 Probleme

Die Implementation, um Meta-Daten abfragen zu können, erwies sich als große Herausforderung. Um Rust-Code zu kompilieren, muss garantiert sein, dass nicht mehrere Threads dieselbe Variable verändern. Die Problematik war, dass die Implementation von Raft nicht die notwendigen Garantien für Multithreading implementiert hatten. Ein weiteres Problem war, dass nicht ersichtlich war was die eleganteste Möglichkeit ist, um auf die Felder des Servers zugreifen zu können.

13.2 Lösung

Damit aus den REST-Routen immer auf die aktuelle Daten zugegriffen werden kann, mussten wir die Felder aus den Objekten in generische Wrapper Types, die Multithreading Garantien besitzen, implementieren. Die in der Diplomarbeit gewählten Wrapper Types sind Arc und RwLock. Arc ist ein Reference Counter, welcher die strong References auf das Objekt überprüft und sobald sein Counter null erreicht, also keine References auf das Objekt vorhanden sind, das Objekt deallokiert. Jedoch können Arcs nur Daten speichern und keine Daten manipulieren. Deswegen mussten wir noch einen weiteren generisch Wrapper

Type verwenden, nämlich ein RwLock (Read-Write-Lock). Das erlaubt, dass mehrere Reader aber nur einen Writer auf eine Variable zugreifen können. Durch *Interior-Mutability* von Rust lässt sich der Inhalt des RwLocks schreiben, trotz "Immutability" von Arc.

13.3 Implementation

Wenn ein Server bzw. Node gestartet wird, wird die notwendige Konfiguration aus der Konfigurationsdatei geladen und die notwendigen Objekte für den RaftServer erzeugt. Danach wird die Funktion *init* aufgerufen, die die Routen erzeugt und den Webserver startet. Die Funktion nimmt dabei Arcs als Parameter, um diese dann an die Routen weiterzugeben.

Das REST-API dient als Schnittstelle und ruft Handler Methoden, die sich als Client ausgeben und das Resultat wieder über das http Protokoll zurücksenden, auf. Unserer HTTP-Schnittstelle ist also ein Proxy und ist eigentlich ein Raft-Client.

13.4 REST-API

Tabelle 1: REST-API Dokumentation

Methode	URL	Beschreibung
GET	/auth/login	Zeigt den derzeit eingeloggten Nutzer an
POST	/auth/login	Loggt Nutzer ein
POST	/auth/logout	Loggt Nutzer aus
GET	/document/:lid/:id	Abfrage eines Dokuments
POST	/document/:lid	Fügt neues Dokument ein
POST	/document/:lid/transaction/:session	Fügt neues Dokument ein während einer Transaktion
DELETE	/document/:lid/:id	Löscht Dokument
DELETE	/document/:lid/:id/transaction/:session	Löscht Dokument während einer Transaktion
PUT	/document/:lid/document/:id	Manipuliert Dokument
PUT	/document/:lid/transaction/:session/:id	Manipuliert Dokument während einer Transaktion
POST	/transaction/begin/:lid	Startet eine neue Transaktion
POST	/transaction/commit/:lid/:session	Committed eine Transaktion
POST	/transaction/rollback/:lid/:session	Rollt eine Transaktion zurück
GET	/meta/log/:lid/documents	Zeigt alle ID in einem Dokument an
GET	/meta/:lid/state/leader	Zeigt die Sicht des Leaders
GET	/meta/:lid/state/candidate	Zeigt die Sicht des Candidates
GET	/meta/:lid/state/follower	Zeigt die Sicht des Followers
GET	/meta/peers	Zeigt alle Peers an

Tabelle 2: Erklärung der Parameter

Tag	Beschreibung
Id	Id des Dokuments in Uuid-Deklaration
lid	ID des Logs in Uuid-Deklaration
session	ID der Transaktion in Uuid-Deklaration

14 MÖGLICHE ERWEITERUNGEN

14.1 Query Language

Derzeit ist die einzige Möglichkeit Dokumente abzufragen, über die DocumentId. Das erfordert eine Überarbeitung des Payloads, denn zurzeit wird der Payload in Bytes gespeichert.

14.2 Multi User Konfiguration

Die jetzige Implementation der Authentifikation unterstützt nur einen festgelegten Benutzer. Das kann ein Sicherheitsproblem darstellen, wenn man die Rechte eines Benutzers nicht einschränken und keine weiteren Benutzer haben kann.

14.3 Client - Prioritäten

Nicht alle Clients sind gleich wichtig. Aus Quality of Service Gründen könnte es für einen Datenbank-Administrator relevant sein, Clients unterschiedliche Prioritäten zu zuteilen.

14.4 SSL - Http Clients

Daten sollten im Transit nicht Plaintext versendet werden. Denn es könnten in manchen PaenkoDocuments sensible Daten gespeichert sein. Aus diesem Grund wäre es sinnvoll für die HTTP-Schnittstelle, SSL zu implementieren.

14.5 SSL - Peers

In den meisten Fällen sind die Knoten nicht lokal, sondern auf mehrere Rechner verteilt, eventuell sogar über die ganze Welt. Nachrichten die über Capnproto serialisiert wurden, sind sie nicht verschlüsselt und damit nicht “sicher”. Denn Nachrichten der Peers werden über gewöhnliche TcpStreams gesendet. Es ist daher empfehlenswert, auch hier SSL zu implementieren.

14.6 Crosslog - Transactions

Wie bereits erwähnt wurde, kann nur eine Transaktion pro Log gestartet werden. Möchte aber ein Anwender Transaktionen über mehrere Logs hinweg machen, geht das leider nicht. Eine interessante Erweiterung wäre es, wenn man auf mehreren

Logs Aktionen durchführen könnte, mit der Möglichkeit, dass wenn etwas schief geht, die Transaktion wieder zurückgerollt werden kann.

14.7 RemoveServer

Derzeit ist nur das dynamische Hinzufügen von Peers zur Laufzeit möglich. Möchte man jedoch einen Node aus dem Cluster entfernen, geht das nicht. Sobald ein Node ausfällt, versuchen die anderen Nodes ihn immer wieder zu erreichen. Eine mögliche Implementation wäre es das Timeout auf mehrere Minuten zu beschränken. Sobald dieses Timeout erreicht wurde, löschen die Peers diesen Peer aus ihrer Liste.

15 TESTING

Die Datenbank wurde über zwei Systeme getestet. Da wir viel an Raft-rs erweitert haben, konnten wir, über die bestehenden Unit-Tests kontrollieren, ob noch die ursprüngliche Funktionalität vorhanden ist. Da aber Unit-Tests seine Grenzen haben, haben wir dann die Funktionalität über Shell-Skripts bzw. Shell-Curl-Skripts getestet. Die zum Testen verwendeten Shell-Skripts finden Sie im *test* Ordner.

16 SICHERHEITSKONZEPTE

Die Sicherheit der Datenbank kann in zwei Teile unterteilt werden: Peer-Security und Client-Security.

Peer-Security wird relevant, wenn neue Peers dem Cluster beitreten wollen. Um nicht jedem den Beitritt zu gestatten, wurden Community Strings implementiert. Das heißt, damit sich Peers miteinander verbinden können, benötigen sie den gleichen Community String. Dieser kann in der Konfiguration konfiguriert werden und ist Plaintext.

Der zweite Teil ist Client-Security, denn nicht jeder Client soll auf den Cluster zu greifen können. Derzeit wird aber nur ein User unterstützt, der explizit in der Konfiguration mit den Zugangsdaten festgelegt sein muss. Jedoch wurde bei der Implementation darauf geachtet, dass die Implementation für zukünftige Erweiterungen offen ist. Die Client-Zugangsdaten bestehen aus Username und Password. Wobei in der Konfiguration, das Passwort gehashed sein muss.

16.1 Implementation

Um ein neues Authentifizierung-Modul zu implementieren, muss lediglich, das Auth-Trait implementiert werden.

```
pub trait Auth: Clone + Debug + Send + 'static {
    /// Generates hash of type T
    fn hash(&self, plain: &str) -> String;

    /// Checks hash and returns whether it was successful or not
    fn compare(&self, hash1: &str, hash2: &str) -> bool;

    /// Checks hash with given username
    fn find(&self, user: &str, hash: &str) -> bool;
}
```

Am Server wird nur noch die *find* Methode, wenn sich der Client das erste Mal verbindet, aufgerufen. Wenn als Resultat true zurückgegeben wurde, war die Verbindung erfolgreich. Die *hash* Methode wird in der Client Library aufgerufen und die *compare* Methode dient zur Vervollständigung der API und sollte von der Methode *find* aufgerufen werden.

```
connection_preamble::id::Which::Client(Ok(client)) => {
      scoped_debug!("received new connection from a client");

      let client_id = try!(ClientId::from_bytes(client.get_id().unwrap()));
      let client_username = client.get_username().unwrap();
      let client_password = client.get_password().unwrap();

      if !self.auth.find(client_username, client_password) {
           scoped_debug!("Wrong username or password");
      } else {
      scoped_debug!("Username and password are okay");
      self.connections[token].set_kind(ConnectionKind::Client(client_id));
      let prev_token = self.client_tokens
            .insert(client_id, token);

      scoped_assert!(prev_token.is_none(),
"{:?}: two clients connected with the same id: {:?}",
                                                  self,
                                                  client_id);
      }
}
```

17 KONFIGURATION

Den Server über die Commandline Parameter zu starten, hat sich als aufwendig erwiesen. Aus diesem Grund lässt sich der Server über Config-Dateien konfigurieren. Die Datenbank bekommt beim Starten als Parameter den Pfad zur Konfigurationsdatei und liest so die Konfiguration aus. Als Format der Config-Dateien wurde *toml* ausgewählt, da dieses auch bei Rust Projekten als Konfigurationsformat Einsatz findet.

So sieht eine Konfiguration aus:

```
[server]
node_id = 1
node_address= "leader:9000"
community_string="test"
binding_addr="0.0.0.0:3000"

[security]
username = "kper"
password = "a665a45920422f9d417e4867efdc4fb8a04a1f3fff1fa07e998e86f7f7a27ae3"

[[peers]]
node_id = 2
node_address = "follower:9001"

[[logs]]
path = "node0_0"
lid = "3d30aa56-98b2-4891-aec5-847cee6e1703"

[[logs]]
path ="node0_1"
lid ="2275a90d-5884-4498-9ba5-91122e2fe875"
```

Abbildung 25: Beispielkonfiguration

Einfache eckige Klammern markieren ein gewöhnliches Feld, wobei doppelte zu einem Array zusammengefasst werden. Im Server-Feld werden relevante Eigenschaften wie Binding-Adresse für die Rest-Schnittstelle und Node-Address für Raft angegeben. In der Abbildung 25 ist die Testkonfiguration von Kevin Per zu sehen. Um die Konfiguration zu erleichtern wurden auf dem Test-System DNS Einträge für *leader* und *follower* verwendet. Im Security-Feld muss der festgelegte Benutzer, mit einem gehashed Passwort, vermerkt sein. In dieser Konfiguration der Abbildung 25 ist nur ein Peer definiert, damit besteht der Raft-Cluster aus 2 Nodes. Außerdem sind zwei Logs definiert. Das Feld *lid* unter *[[logs]]* dient zur Identifizierung und muss dem Uuid-Standard entsprechen. Des Weiteren ist ein

Path im Log definiert. In diesem speichern das Log und Statemachine Informationen, damit wenn ein Node abstürzen sollte, Informationen persistent gespeichert sind.

18 CLI-REFERENZ

Neben der REST-API lässt sich die Datenbank auch über Command-Line steuern, denn beide basieren gleichermaßen auf demselben Handler. Außerdem, sind die Befehle für Client und Server in derselben Binary kompiliert. Für die genaue Anwendung der Befehle finden sich weitere Beispiele im test Verzeichnis im Projekt.

```
document get <doc-id> <lid> <node-address> <username> <password>
document put <doc-id> <lid> <node-address> <filepath> <username> <password>
document post <lid> <node-address> <filepath> <username> <password>
document remove <doc-id> <lid> <node-address> <username> <password>
document server  <config-path>
document begintrans <lid> <node-address> <username> <password>
document commit <lid> <node-address> <username> <password> <transid>
document rollback <lid> <node-address> <username> <password> <transid>
document transpost <lid> <node-address> <filepath> <username> <password> <transid>
document transremove <lid> <node-address> <doc-id> <username> <password> <transid>
document transput <lid> <node-address> <doc-id> <filepath> <username> <password> <transid>
```

Abbildung 26: CLI Referenz

Tabelle 3: Definition der CLI Parameter

Tag	Beschreibung
doc-id	Id des Dokuments in Uuid-Deklaration
lid	ID des Logs in Uuid-Deklaration
node-address	IP-Adresse und Port vom Node
username	Username vom Benutzer
password	Passwort vom Benutzer
filepath	Pfad der Datei, die hochgeladen werden soll
trans-id	ID der Transaktion in Uuid-Deklaration
config-path	Pfad der Config-Datei

19 User-Handbuch

19.1 Getting Started

Zuerst benötigen Sie das Projekt. Das Projekt ist öffentlich über Github herunterladbar.

```
git clone https://github.com/paenko/paenkodb
```

Um die Datenbank kompilieren zu können, benötigt man den Rust-Compiler und den Dependency Manager von Rust, Cargo. Beides lässt sich sehr einfach mittels Rustup installieren.

```
curl -L https://static.rust-lang.org/rustup.sh > rustup
chmod +x rustup
./rustup --channel=nightly
```

Außerdem empfehlen wir den Nightly Compiler, da einige Features, die benutzt wurden, noch nicht im Stable Build von Rustc sind.

Die Kommunikation zwischen den einzelnen Peers erfolgt über Capnproto. In dem messages.capnp sind alle Messageformate definiert. Diese müssen aber erst in Rust-Code übersetzt werden. Dazu benötigen Sie Capnproto. Dieses Programm wird automatisch beim Builden der Datebank aufgerufen und muss auf Ihrem System installiert sein.

```
git clone https://github.com/sandstorm-io/capnproto.git
cd capnproto/c++
./setup-autotools.sh
autoreconf -i
./configure
make -j6 check
sudo make install
```

Einige Dependencies vom Projekt benötigen Systembibliotheken. Deswegen werden Openssl und eventuell gewisse andere Libraries auf dem Server vorausgesetzt. Falls diesbezüglich Fehler auftreten sollten, befolgen Sie am besten die Anweisung auf https://github.com/sfackler/rust-openssl.

Nachdem alle Dependencies installiert sind, wechseln Sie in das Verzeichnis *src/document* und kompilieren das Projekt mit *cargo build*. Das sollte weitere Dependencies automatisch für das Projekt installieren und gleich danach auch kompilieren. Die Binary finden Sie dann Verzeichnis *target/debug*. Wenn Sie nun in dieses Verzeichnis wechseln, können Sie mit *./document server config.toml* einen Server starten. Die config.toml können Sie aus *docker/config* kopieren. Sie müssen jedoch die Konfiguration anpassen, indem Sie die IP-Adressen und Logs ändern, da die vorkonfigurierten Werte eigentlich für Docker gedacht waren. Sie können die Konfiguration so anpassen, dass ihr Cluster auch aus so viele Nodes besteht, wie es Ihnen beliebt.

19.2 Docker

Uns war die lange Liste an Dependencies bewusst und um das Kompilieren zu vereinfachen, lässt sich das Projekt auch über Docker “bauen”. Dazu wechseln Sie in das Docker Verzeichnis und je nachdem welchen Branch Sie kompilieren möchten, müssen Sie das gewünschte Dockerfile verwenden. Dazu können Sie zwischen Dockerfile für den master-Branch und Dockerfile_dev für den dev-Branch wählen.

Master:

```
docker build -t paenkodb .
```

Dev:

```
docker build -t paenkodb:dev -f Dockerfile_dev .
```

Sobald der Build erfolgreich war, können Sie über *docker-compose up*, die von uns erstellte Standard Konfiguration verwenden. Sie können die *data* Verzeichnisse in der Config bzw. in docker-compose.yml ändern.

20 C# LIBRARY

Die PaenkoDb C# Library ist eine Klassenbibliothek die Entwicklern eine Schnittstelle zur Verfügung stellt, die es ermöglicht, einfach und ohne Vorkenntnisse von PaenkoDB mit der Paenko Datenbank zu arbeiten. Die Bibliothek bietet folgende Features.

- Das Vornehmen von Create, Read, Update und Delete Operationen an der Datenbank über die File-ID.
- Geographische Lokalisierung von Paenko Datenbanknodes.
- Health Checks die überprüfen ob ein Datenbanknode ausgefallen ist.

20.1 CRUD Operationen

Die Daten der Datenbank werden über eine Rest Schnittstelle übertragen, die Library übernimmt über HTTP-Requests die Kommunikation mit dieser Schnittstelle um den Datentransfer transparent zu halten. Um einen CRUD Befehl ausführen zu können muss ein Username und ein Passwort angegeben werden, diese Daten werden am Server überprüft und in einer HTTP Session gehalten.

Die für die Kommunikation verwendeten HTTP-Request können sowohl synchron als auch asynchron aufgerufen werden was bei der Verwendung der Library innerhalb eines GUIs dafür sorgt, dass dieses responsive bleibt.

CRUD Befehle der Library können auch als Teil einer laufenden Transaktion ausgeführt werden. Eine Transaktion kann mit Hilfe der Library begonnen, geschlossen und zurückgerollt werden.

20.2 Geotracking

Beim Hinzufügen eines Paenko Datenbanknodes durch die Library kann der User entscheiden ob er seinen Node mithilfe von http://www.ip-api.com/ geographisch lokalisieren möchte. Dabei werden Informationen wie Längengrad, Breitengrad und Zeitzone im Node Objekt gespeichert und können zu einer späteren Zeit zum visualisieren des Nodes verwendet werden. Falls diese Funktion nicht verwendet wird, sind die Daten bei einer Abfrage “UNKNOWN”.

20.3 Healthchecks

Da die Paenko Datenbank auf Ausfallsicherheit ausgelegt ist, stellen wir in der von uns entwickelten C# Library Funktionen zur Verfügung, die überprüfen ob ein oder mehrere Datenbanknodes erreichbar sind.

Diese Funktionen können entweder einmalig oder in einem bestimmten Intervall ausgeführt werden, so kann zum Beispiel alle 10 Minuten auf Ausfälle im System geprüft werden. Nach jedem vergangenen Intervall wird hierbei eine Callback Methode ausgeführt. In dieser Callback Methode kann der User dafür sorgen, dass er seine Nodes wieder funktionsfähig macht.

20.4 Anwendung

Im folgenden Abschnitt werden die von außen ausführbaren Funktionen der Library erläutert. Die Library beinhaltet folgende Klassen (Klassendiagramm):

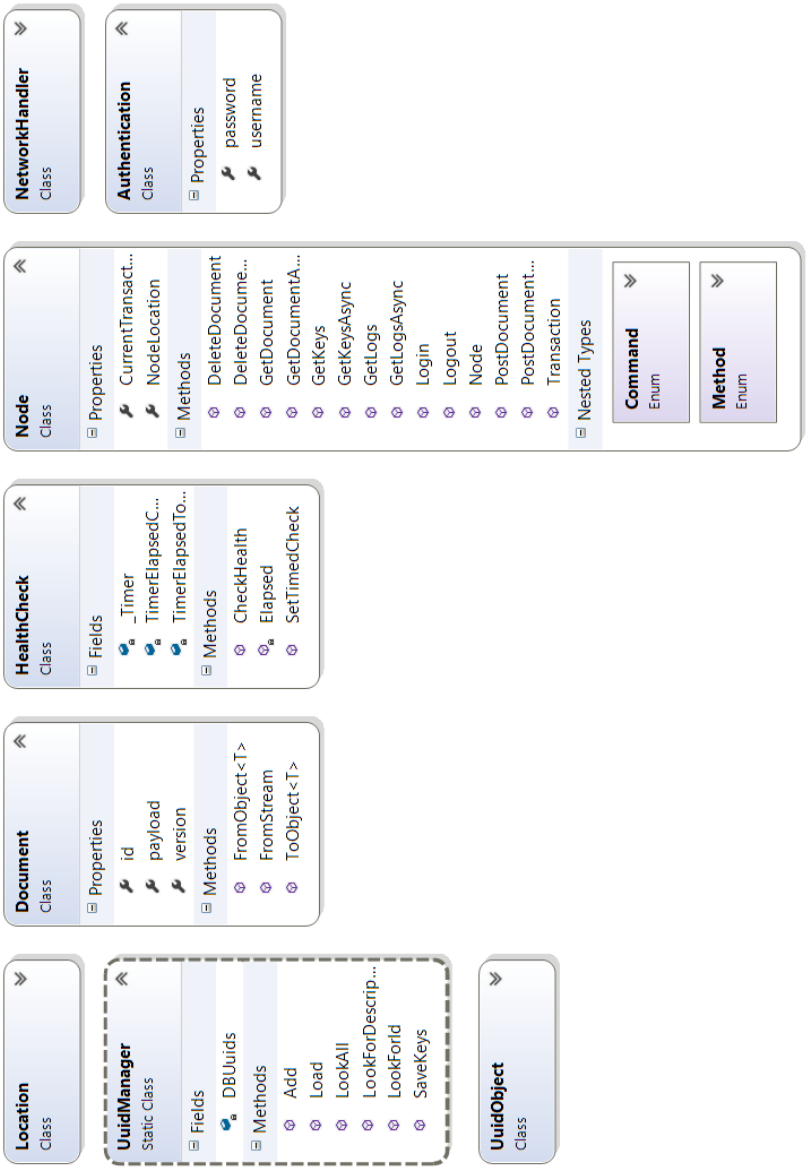


Abbildung 7: C# Library Klassendiagramm

20.5 Node

Die meiste Arbeit der Library läuft in der Node Klasse ab. Diese Klasse definiert den Datenbank-Server auf dem gearbeitet wird und führt alle CRUD befehle aus.

```
public void Login(Authentication auth)
public void Logout()
```

Vor der Anwendung von Datenbank-Befehlen muss sich der Benutzer der Library mit seinem Usernamen und Passwort aus dem Konfigurationsfile der Datenbank anmelden. Durch diese Anmeldung wird der Benutzer authentifiziert und es wird eine Session für alle folgenden Befehle gestartet. Der User kann dann in dieser Session arbeiten ohne sich erneut authentifizieren zu müssen. Mit der Funktion zum Ausloggen wird diese Session wieder beendet, nachdem die Session geschlossen ist können keine Datenbank-Befehle ausgeführt werden.

```
public Document GetDocument(UuidObject log, UuidObject file)
public string DeleteDocument(UuidObject log, UuidObject file)
```

Mit diesen beiden Befehlen können Dokumente auf der Datenbank heruntergeladen oder gelöscht werden. Die UuidObject Parameter stehen jeweils für den Log und das File mit dem gearbeitet werden soll. UuidObjects werden über den UuidManager organisiert.

```
public void PostDocument(Document doc, UuidObject log, string addedUuidDescription, Method method = Method.Post)
```

Um ein Dokument an die Datenbank zu senden verwendet man PostDocument, diese Funktion verlangt das Dokument das gesendet werden soll, den Log auf den gearbeitet wird, eine Beschreibung des Dokuments für späteren Aufruf und optional eine Sende-Methode. Bei der Sende-Methode handelt es sich um die Art wie ein Dokument abgespeichert werden soll, hierfür gibt es zwei Möglichkeiten, die Methode Post um ein neues eigenständiges Dokument zu senden und Put um ein vorhandenes zu überschreiben. Falls der User ein Dokument überschreiben möchte muss er die Id des Dokuments welches er überschreiben möchte vor der Ausführung der PostDocument Methode im Dokument Objekt festlegen.

```
public List<string> GetLogs()
public List<string> GetKeys(UuidObject log)
```

Über die Methode GetLogs kann der Benutzer der Library alle Logs abfragen die zurzeit am Server in Betrieb sind. Um alle File-Ids eines bestimmten Logs zu erfahren kann die Methode GetKeys mit dem log als Parameter verwendet werden.

```
public void Transaction(UuidObject log, Command command)
```

Die Transaction Methode ermöglicht das Arbeiten mit Transaktionen über die Library. Ein Transaktions Befehl wir auf einem bestimmten Log mit einem Kommando ausgeführt, dieses Kommando ist entweder begin, commit oder rollback. Falls eine neue Transaktion begonnen wird, werden alle folgenden CRUD-Befehle als Teil dieser Transaktion gesendet.

20.5.1 Document

```
public static PaenkoDocument FromObject<T>(T docobj)
public static PaenkoDocument FromStream(Stream docstream)
```

Ein zu sendendes Dokument kann entweder über ein Objekt oder einen Stream erzeugt werden. Das bedeutet, dass ein Benutzer sowohl eine ganze Datei von seinem Rechner, als auch serialisierte Zustände über die Library auf der Datenbank abspeichern kann. Diese serialisierten Zustände können nach dem Herunterladen von der Datenbank natürlich wieder deserialisiert werden.

20.5.2 UuidManager

```
public static UuidObject LookForId(string description)
```

Um das Arbeiten mit File-Ids und Log-Ids zu vereinfachen haben wir eine Klasse implementiert welche die Ids organisiert. Ids die von der Datenbank erhalten werden können mit einer Beschreibung versehen und danach mit Hilfe des Managers im Kontext von CRUD-Befehlen über diese Beschreibung aufgerufen werden. Die Ids zusammen mit den Beschreibungen werden lokal im JSON Format gespeichert.

```
public static void Load(string file)
public static void SaveIds(string file)
```

Zu Beginn des Programms müssen die gespeicherten Ids geladen werden, nachdem sie geändert wurden speichert die SaveIds Funktion die Ids wieder ab.

20.5.3 Benutzungsbeispiel

Ein Beispielprogramm das unsere Library verwendet und testet:

```
//Verbindung zum Datenbank-Server über das Node Objekt herstellen
Node node = new Node(IPAddress.Parse("207.154.216.94"), 3000);
//Authentifizierung mit Username und Passwort
node.Login(new Authentication("ich", "pw"));
//Laden der Uuids welche lokal gespeichert sind
UuidManager.Load("ids.json");
//Alle Logs vom Server erhalten
var logs = node.GetLogs();
//Diese Logs ausgeben
logs.ForEach(l => Console.WriteLine($"log: {l}"));
//Wenn es im UuidManager keine Definition für HauptLog gibt wird diese erstellt,
//sonst wird dieser Schritt übersprungen.
if (UuidManager.LookForId("HauptLog") == null)
{
    UuidManager.Add(logs[0], "HauptLog", UuidObject.UuidType.Log);
}
//Bankdaten erzeugen
Bankdaten tk = new Bankdaten();
//Diese Bankdaten in ein Document verpacken damit sie an den Server geschickt werden können
Document x = Document.FromObject<Bankdaten>(tk);
//Wenn es im UuidManager keine Definition für Bankdaten gibt ist klar, dass das File noch
//nie an den Server geschickt wurde, deshalb posten wir es. Wenn eine Definition existiert
//wird dieser Schritt übersprungen.
if (UuidManager.LookForId("Bankdaten") == null)
{
    node.PostDocument(x, UuidManager.LookForId("HauptLog"), "Bankdaten");
}
//Bankdaten über die Uuids herunterladen
var response = node.GetDocument(UuidManager.LookForId("mainlog"),
    UuidManager.LookForId("TestClassFile"));
//Bankdaten deserialisieren
var bankdaten = response.ToObject<Bankdaten>();
//Bankdaten ausgeben
Console.WriteLine($"a: {bankdaten.Konto1} b: {bankdaten.Konto2}");
//Mögliche Änderungen an den Ids wieder lokal abspeichern
UuidManager.SaveIds("ids.json");
```

Abbildung 8: C# Library Beispielprogramm

21 C# MANAGER

Der Paenko C# Manager ist ein Management Tool das dabei hilft Paenko Datenbanknodes zu visualisieren und zu verwalten. Beim Start werden die Node Daten aus einem File ausgelesen und nach einem bestätigen der Node Liste werden diese Nodes auf einer Landkarte eingetragen auf der man die Daten der einzelnen Nodes einsehen und manipulieren kann.

Die Manipulation der Daten geschieht ganz einfach über Drag&Drop in das File-Fenster des jeweiligen Nodes. Wenn eine Datei in das Fenster hineingezogen wird, wird es entweder als neues File gesendet oder wenn die PUT Option aktiviert ist upgedatet, nachdem die Fileid des Files das zu überschreiben ist angegeben wird. Das Programm ermöglicht auch das starten und beenden von Transaktionen.

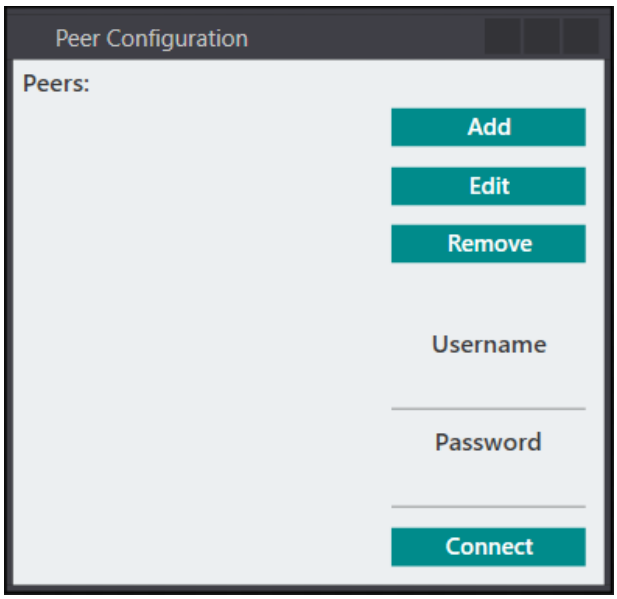


Abbildung 9: C# Manager Konfigurationsfenster

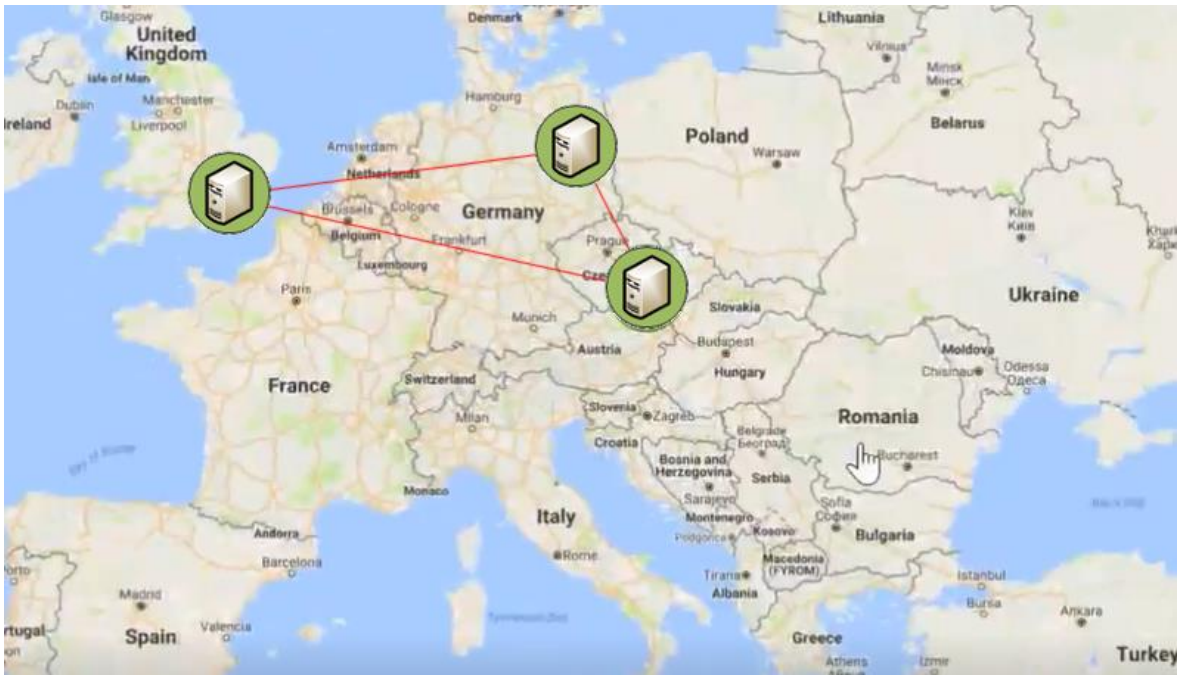


Abbildung 10: C# Manager Google Map

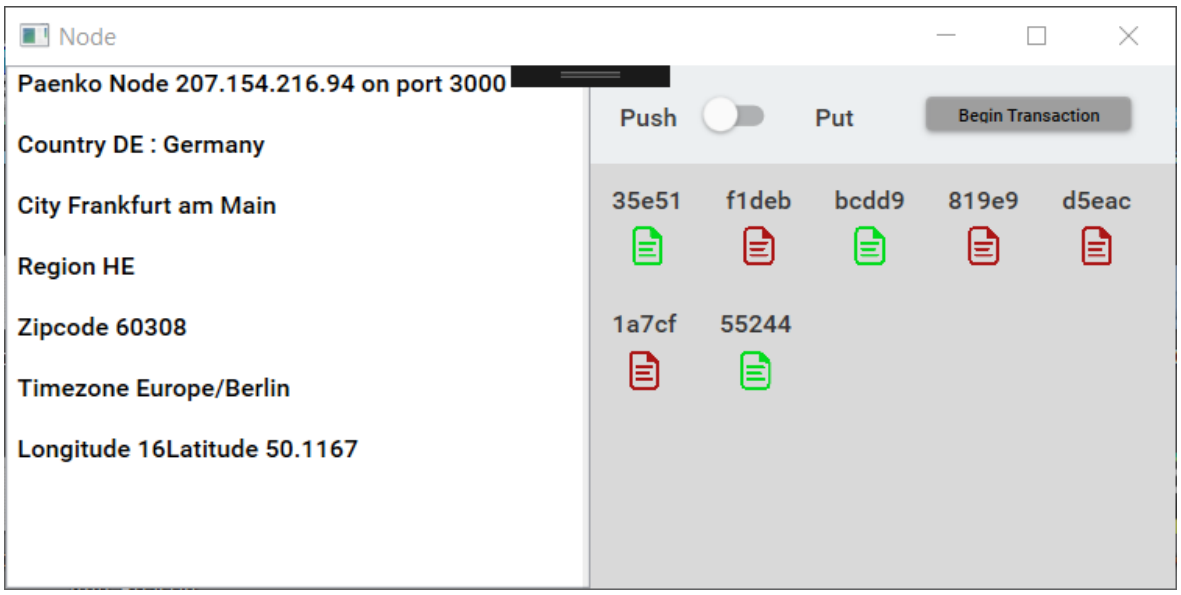


Abbildung 11: C# Manager Drag & Drop

21.1 Entwicklung

Um die Google Maps Karte in der WPF Anwendung darzustellen haben wir den C# Wrapper von Mike Hall verwendet. Dieser Wrapper ermöglicht es uns eine Karte zu generieren und auf dieser anklickbare Marker einzuzeichnen. Im Programm sind Nodes die erreichbar sind in grün eingezeichnet und die, die es nicht sind in Rot.

Der Wrapper funktioniert zwar für unseren Anwendungszweck, ist jedoch etwas veraltet, da er aus dem Jahr 2015 ist. Das könnte für ein Problem für zukünftige Erweiterungen des Programms darstellen weil moderne Funktionen des Google Maps API nicht unterstützt sind.

22 C# WEBINTERFACE

Auf jedem PaenkoDB Datenbank-Node läuft ein HTTP Server mit dem Paenko Webinterface. Das Interface hilft genauso wie das C# Manager Programm beim Organisieren einer Paenko Datenbank. Es basiert auf Angular 2 und verwendet das REST Interface der Datenbank-Server um Daten der Datenbank auszulesen. Da auch die C# Library das REST Interface verwendet um die Daten zu manipulieren sind sich Webinterface und C# Library in der Funktionsweise etwas ähnlich.

Da wir für das Design der Seite bootstrap verwendet haben ist sie auf verschiedenen Geräten responsive.

22.1 Funktion

Über die Webapplikation können die Daten, die sich am Server befinden, und eine Liste aller Nodes im System eingesehen werden. Das hilft dem User dabei die Konfiguration der Datenbank zu visualisieren und somit zu vereinfachen.

Falls ein neuer Node zum System hinzugefügt werden soll kann der User dafür ebenfalls das Webinterface verwenden. Hierfür muss lediglich die IP-Adresse des Nodes angegeben werden.

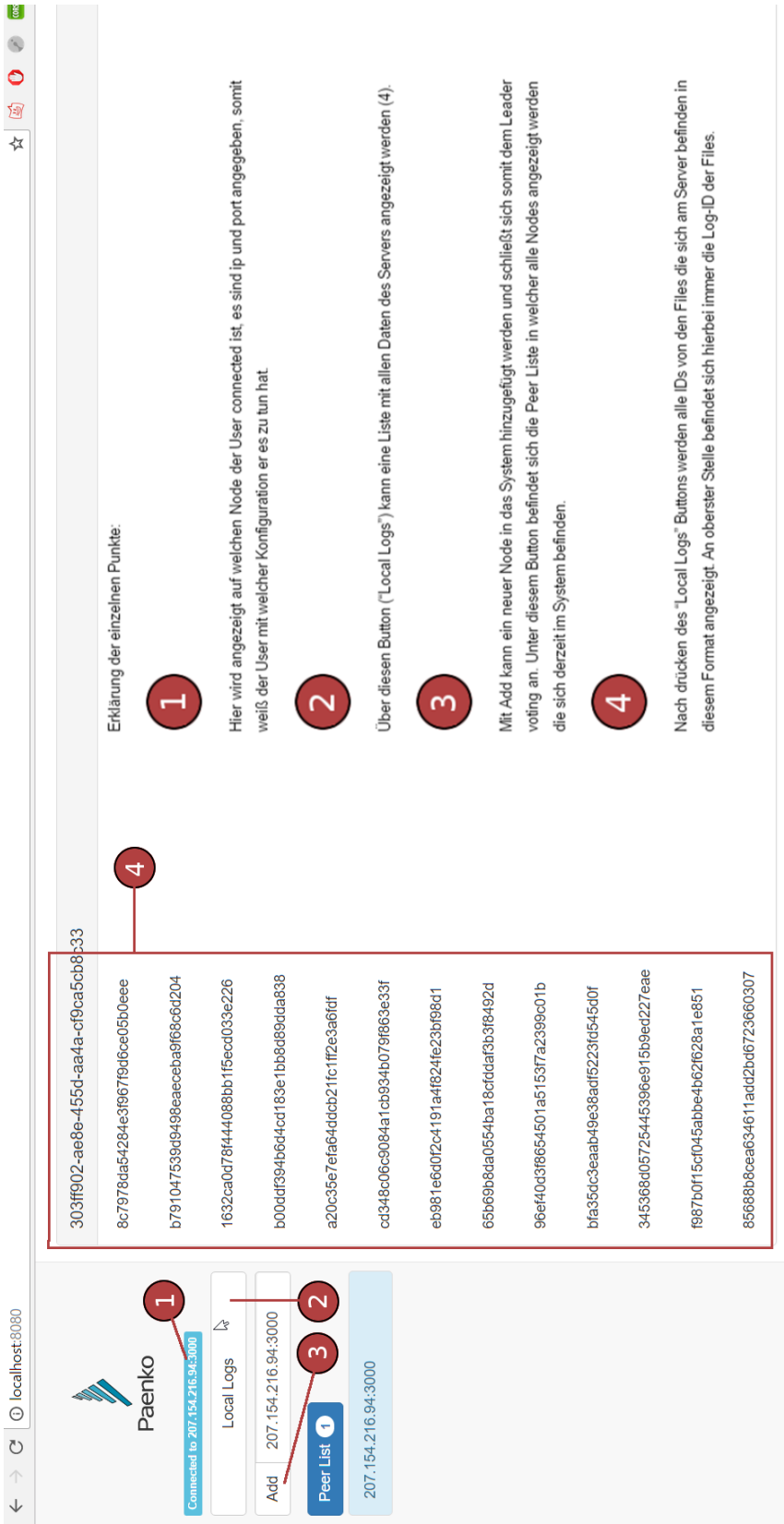


Abbildung 12: Webinterface mit Erklärung

23 MÖGLICHE FRONTEND-ERWEITERUNGEN

23.1 Datenbank-Treiber

Eine mögliche Erweiterung der Datenbank sind Datenbank Treiber in verschiedenen Programmiersprachen. Wir haben uns beim Frontend auf die Programmiersprache C# fokussiert, da wir diese in der Schule gelernt haben und deshalb sehr gut beherrschen aber Programme wie die C# Library sind für das Arbeiten mit der Datenbank sehr wichtig sodass sie auch in anderen Programmiersprachen eine gute Erweiterung für PaenkoDB sind.

Das Entwickeln eines solchen Treibers sollte keine allzu große Herausforderung darstellen, da die Datenbank über eine REST-Schnittstelle kommuniziert und somit leicht einzubinden ist.

24 WETTBEWERBE UND FÖRDERUNGEN

Wir haben unser Projekt PaenkoDB bei verschiedenen Wettbewerben und Förderungen eingereicht und waren dabei sehr erfolgreich.

24.1 ITS Award

Wir haben unser Projekt beim 13. jährlichen ITS Award der FH Salzburg eingereicht und zählen nun zu den 10 Finalisten die aus 30 Einreichungen ausgewählt wurden.

Wir waren am 10. März 2017 bei der Kick-off-Veranstaltung in Salzburg und haben dort die anderen Finalisten kennengelernt sowie unsere Startförderung in Höhe von 200€ erhalten.

Das Finale findet am 30. Mai 2017 statt und wir sind uns sicher, dass wir eine gute Bewertung erhalten werden.

24.2 AXAWARD

Am 3. März 2017 haben wir uns für den Austrian X.TEST AWARD angemeldet. Dieses Event ist bekannt als Talent-Wettbewerb für junge technikbegeisterte Schüler und Schülerinnen und wird heuer zum sechsten Mal ausgeschrieben.

Am 12. April 2017 erfahren wir, ob auch wir unter den 10 Finalisten stehen.

24.3 Jugend Innovativ

“Jugend Innovativ ist der größte österreichische Schulwettbewerb für innovative Ideen. Er wird im Auftrag des Bundesministeriums für Wissenschaft, Forschung und Wirtschaft sowie des Bundesministeriums für Bildung von der Austria Wirtschaftsservice GmbH abgewickelt und von der Raiffeisen Klimaschutz-Initiative unterstützt. Der Wettbewerb wird laufend von Weiterbildungs-Maßnahmen für Lehrerinnen und Lehrer zu den Themen „Teaching Innovation“ und „Rechte an geistigem Eigentum für Schulprojekte“ sowie von Praxis-Workshops für Schülerinnen und Schüler erfolgreich begleitet.” - http://www.jugendinnovativ.at/News/5312.html

Die Einreichung bei Jugend innovativ hat uns den meisten Aufwand bereitet, da wir nach der Anmeldung eine ausführliche schriftliche Ausarbeitung erstellen mussten. Das Finale wird vom 31. Mai bis 2. Juni stattfinden.

24.4 Netidee Förderung

Netidee ist eine Organisation, die finanziert von der Internet Foundation Austria, Projekte und Aktivitäten unterstützt, die die weitere Verbreitung und vielseitige Nutzung des Internet in Österreich fördern. Seit 2006 findet jedes Jahr ein netidee-Call statt, dieses Jahr wurden 28 Projekte und 8 Stipendien von insgesamt 165 Einreichungen gefördert, darunter auch PaenkoDB.

Am 17. November 2016 waren wir beim netidee best of 2016 wo alle geförderten Ideen vorgestellt wurden. Die gesamte Förderung beträgt 3000€, mit dem Geld haben wir uns Hardware gekauft, die wir bei der Diplomarbeit benötigt haben.

25 Soziale Netzwerke

25.1 Facebook

Wir haben einige der Meilensteine auf Facebook festgehalten, zum Beispiel den Besuch an der FH Salzburg zum Kick-Off-Meeting des ITS Awards und unseren Auftritt am Tag der Offenen Tür 2017 der HTL Ottakring.

25.2 Trello

Um immer auf dem Laufenden zu bleiben haben wir uns auf der Seite www.Trello.com einen Backlog erstellt in dem wir unsere Aufgaben und Todos eingetragen haben. Somit wussten wir immer welche Aufgaben erfüllt waren und wie weit der jeweils andere mit seiner Arbeit ist.

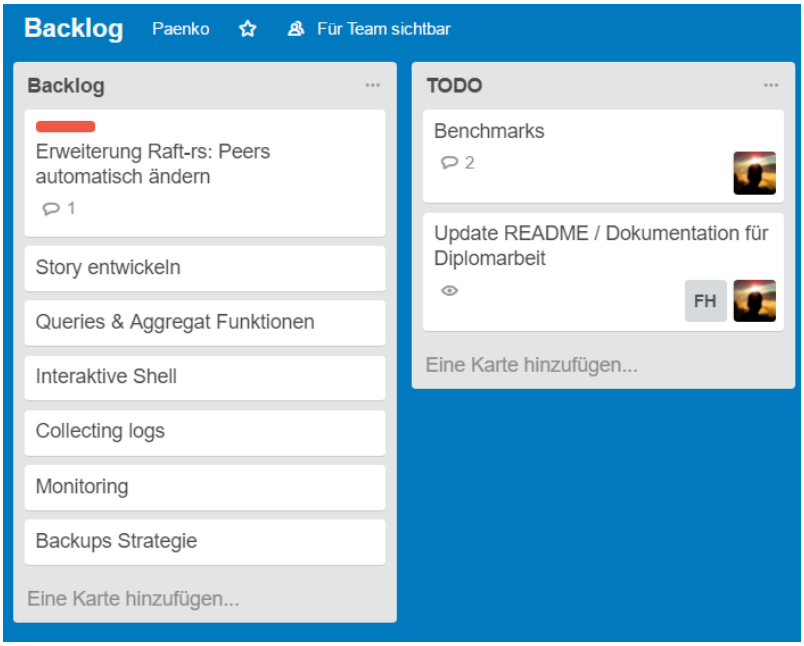


Abbildung 13: Trello Backlog

25.3 Github

Uns war von Anfang an klar, dass PaenkoDB ein Opensource-Projekt werden sollte. Nicht nur weil wir dadurch Feedback der Community erhalten, sondern auch gleichzeitig ein bisschen Werbung für die Datenbank betreiben.

Derzeit umfasst die Diplomarbeit vier Repositories:

1. PaenkoDb
2. PaenkoDB_CSharp
3. PaenkoDB_CSharp_Client
4. PaenkoDB_Web

Alle Repositories sind zu finden unter dem Link https://www.github.com/paenko.

Um uns gegenseitig Feedback zu unserer Arbeit zu geben haben wir auf Github Issues geschrieben die mögliche Inkonsistenzen oder fehlende Features beschreiben. Ein Beispiel für ein solches Issue:

Abbildung 14: Github Issue Beispiel

26 LESSONS LEARNED

Im Zuge des Projektes haben wir sehr viel über die Zusammenarbeit im Team gelernt. Wir haben gelernt wie wichtig es ist konstruktives Feedback zu geben und sich laufend klare Ziele zu setzen. Uns ist aufgefallen, dass textbasierte Kommunikation auch mit den besten Tools nicht immer genügt um sich miteinander auszutauschen, aber definitiv sehr viel bringt.

Ein so großes Projekt auf die Beine zu stellen war für beide von uns natürlich eine große Herausforderung, doch wir haben uns selbst bewiesen, dass es, wenn man mit Ehrgeiz an die Sache herangeht, wenige Probleme gibt die nicht zu lösen sind.

Selbstverständlich haben wir auch sehr viel über Datenbanken und die Theorie hinter ihnen gelernt, was uns in unserem späteren Berufsleben sicherlich weiterhilft.

VERZEICHNIS DER TABELLEN

Tabelle 1: REST-API Dokumentation 37

Tabelle 2: Erklärung der Parameter..... 38

VERZEICHNIS DER ABBILDUNGEN

Abbildung 1: Google Spanner Architektur 14

Abbildung 2: Architektur Diagramm von PaenkoDb(links) und Frontend(rechts) 15

Abbildung 3: Ablauf der States in Raft..... 17

Abbildung 4: Überblick von PaenkoDb 21

Abbildung 5: Ablauf einer Transaktion..... 25

Abbildung 6: Ablauf von ServerAdd..... 29

Abbildung 7: C# Library Klassendiagramm 51

Abbildung 8: C# Library Beispielprogramm..... 54

Abbildung 9: C# Manager Konfigurationsfenster..... 55

Abbildung 10: C# Manager Google Map 56

Abbildung 11: C# Manager Drag & Drop..... 56

Abbildung 12: Webinterface mit Erklärung..... 59

Abbildung 13: Trello Backlog..... 63

Abbildung 14: Github Issue Beispiel 64

ANHANG

Arbeitsaufteilung

Person	**Folgende Punkte des Diplomarbeitshandbuches wurden von folgenden Personen geschrieben, inklusive aller Unterpunkte**

Kevin Per	- Das Paenko Ecosystem
	- Konsensalgorithmus Raft
	  ○ Statemachine und Log
	  ○ Rollenverteilung
	  ○ Fehlertoleranz durch Timeouts
	  ○ Kommunikation
	  ○ Log Konsistenz
	- Implementation von Raft-rs
	  ○ StatemachineStatemachine - apply() und query()
	  ○ Log
	- Transaktionen
	  ○ Atomicity
	  ○ Consistency
	  ○ Isolation
	  ○ Durability
	  ○ Kommunikation
	  ○ Implementation
	- Dynamic Peer Adding
	- Multi-Log
	  ○ Implementation
	- Transparenz
	- Kommunikation und REST-Schnittstelle

	○ Probleme ○ Lösung ○ Implementation ○ REST-API - Mögliche Erweiterungen ○ Query Language ○ Multi User Konfiguration ○ Client - Prioritäten ○ SSL - Http Clients ○ SSL - Peers ○ Crosslog - Transactions - Testing - Sicherheitskonzepte ○ Implementation - Konfiguration ○ CLI-Referenz - User-Handbuch ○ Getting Started ○ Docker
Florain Hanko	- Kurzfassung - Abstract - Zielsetzung ○ Hauptziele ▪ Entwickeln der verteilten Datenbank ▪ Kommunikation zwischen Datenbank-Nodes ▪ Entwickeln des User Interface ▪ C# Verwaltungssoftware ▪ Entwickeln einer User Authentifizierung

- Anwendungsszenarien
  - Anwendung im Gesundheitsbereich
  - **Fehler! Verweisquelle konnte nicht gefunden werden.**
- Konkurrenzanalyse
  - Apollo von Facebook
  - Spanner von Google
- C# Library
  - CRUD Operationen
  - Geotracking
  - Healthchecks
  - Anwendung
  - Node
    - Document
    - UuidManager
    - Benutzungsbeispiel
- C# Manager
  - Entwicklung
- C# Webinterface
  - Funktion
- Mögliche Frontend-Erweiterungen
  - Datenbank-Treiber
- Wettbewerbe und Förderungen
  - ITS Award
  - AXAWARD
  - Jugend Innovativ
  - Netidee Förderung
- Soziale Netzwerke
  - Facebook
  - Trello
  - Github
- Lessons Learned

Arbeitsprotokoll – Kevin Per

September

Im September wurden erste Gedanken über die Architektur von PaenkoDb gemacht. Außerdem wurden die ersten Interfaces von raft-rs implementiert.

Oktober

Die erste Version der Webseite wurde entwickelt und erste REST-Routen wurden erstellt. Außerdem wurde Peer-Security (Community Strings) entwickelt. Es wurde zusätzlich die erste Version der Authentifikation entwickelt.

November

Zwischen Oktober und November wurde an Transaktionen gearbeitet. Ende November wurden noch Bugs gefixed. Außerdem wurde der LogManager für das MultiLog Feature implementiert.

Dezember

Im Dezember wurde hauptsächlich an Multilog gearbeitet. Außerdem wurde mit Hilfe von “clippy” der Code optimiert.

Jänner

Im Jänner wurde die Statemachine refactored und Meta-Rest-Routen wurden implementiert. Außerdem wurde die Serialisierungs und Deserialisierungslibrary von “rustc-serialize” auf “serde” geändert.

Februar

Im Februar wurde “dead” Code entfernt und es wurde an einem Docker-Deployment gearbeitet. Zusätzlich wurden neue Http-Routen geschrieben, damit man Transaktion auch über das REST-Interface verwenden kann. Außerdem wurde das “dynamic peering” Feature implementiert.

März

Im März wurde die Statemachine wieder refactored. Des Weiteren wurde an “SimpleAuthentication” gearbeitet. Außerdem wurden die Transaktionen

refactored, damit wenn Fehler auftreten, passende Fehlermeldung ausgegeben werden.

Zusätzlich wurde die Konfiguration für Docker fertiggestellt. Es wurden auch http sessions implementiert, damit wenn ein User sich eingeloggt hat, die Session weiter besteht.

Des Weiteren wurde am Diplomarbeitsbuch geschrieben.

Arbeitsprotokoll – Florian Hanko

November

Init Commit der C# Library. Mit diesem Commit kann der User der Library REST Anfragen an Datenbankserver schicken und sie über die Library pingen um zu sehen ob sie gerade in Betrieb sind.

Ein weiterer Commit der Library erlaubt es die Positionen der Server abzurufen und diese zu speichern um im Verlauf des Programms mit diesen Informationen zu arbeiten. Init Commit des C# Managers. Datenbankserver können über den Manager manipuliert werden.

Am 19. November wurde eine Google Maps library in den Manager eingebaut um Datenbankserver zu visualisieren.

Dezember

Es wurden asynchrone Methoden in der C# Library implementiert welche dieselben Aufgaben erfüllen wie die synchronen REST Methoden.

Der C# Manager kann nun über Drag&Drop File senden und erhalten.

Jänner

Über die neuen zeitbasierten Healthcheck Methoden der C# Library, kann der User nun in einem bestimmten Intervall prüfen ob seine Server in Betrieb sind.

Februar

Die Library wurde auf die neuen Routen des Datenbank-Rest-Servers angepasst und es wurden Funktionen implementiert die das starten von Transaktionen über die Library erlauben. Außerdem enthält die Library nun Funktionen die es ermöglichen ein Objekt oder einen Stream zu serialisieren und so an die Datenbank zu senden. Von der Datenbank erhaltene Objekte können natürlich auch deserialisiert werden.

Die C# Library erhielt am 21.2.2017 ein kleines refactoring der asynchronen Methoden.

Der C# Manager wurde an die neue Library version angepasst.

Init Commit des Paenko Webinterfaces. Über das Webinterface können nun Daten der Datenbankserver eingesehen werden.

März

Es wurde ein riesiges refactoring der C# Library durchgeführt um den Code nachvollziehbarer zu machen.

Der C# Manager wurde an die neue Library version angepasst.

April

Es wurde ein Nuget Paket für die C# Library erstellt.

Die Peeranzeige des Webinterfaces wurde überarbeitet.

Diplomarbeitsantrag

HÖHERE TECHNISCHE BUNDESLEHRANSTALT WIEN 16, THALIASTRASSE 125
Höhere Abteilung für Informationstechnologie
Ausbildungsschwerpunkt Netzwerktechnik und Medientechnik

Erklärung

Die Kandidaten / Kandidatinnen nehmen zur Kenntnis, dass die Diplomarbeit in eigenständiger Weise und außerhalb des Unterrichtes zu bearbeiten und anzufertigen ist, wobei Ergebnisse des Unterrichtes – als solche klar gekennzeichnet – mit einbezogen werden können.
Die Abgabe der vollständigen Diplomarbeit hat bis spätestens

4.4.2017

beim zuständigen Prüfer / der zuständigen Prüferin in ausgedruckter (2 Exemplare) und digitaler
Form (CD-ROM, DVD)zu erfolgen.

Kandidaten / Kandidatinnen	Unterschrift
Kevin Per	
Florian Hanko	

OStR Ing. DI Robert
Baumgartner MBA
Betreuer/in

Mag. Thomas Angerer
Abteilungsvorstand

DI Peter Johannes Bachmair
Direktor

Genehmigung

Wien, am _____

LSI HR DI Judith Wessely-Kirschke

Inhaltsverzeichnis

1 PROJEKTIDEE

1.1 Ausgangssituation

1.2 Beschreibung der Idee

2 PROJEKTZIELE

2.1 MUSS Ziele

2.2 Optionale Ziele (Soll, Kann Ziele)

2.3 NICHT Ziele

3 PROJEKTORGANISATION

3.1 Grafische Darstellung (Organigramm)

3.2 Projektteam

3.3 Individuelle Aufgabenstellung

4 PROJEKTUMWELTANALYSE

4.1 Grafische Darstellung

4.2 Beschreibung der wichtigsten Umwelten

5 RISIKOANALYSE

5.1 Beschreibung der wichtigsten Risiken

5.2 Risikoportfolio

5.3 Risiko Gegenmaßnahmen

6 MEILENSTEINLISTE

7 KOSTENABSCHÄTZUNG

7.1 Finanzierung

8 MOTIVATION

8.1 Kevin Per

8.2 Florian Hanko

1 Projektidee

1.1 Ausgangssituation

Im Zeitalter des Internets elektrisiert kaum ein Thema mehr als die verlässliche Speicherung und Auswertung von großen Datenmengen. Denn täglich fallen immer mehr Daten an, die verarbeitet werden müssen. Besonders in großen und mittleren Unternehmen ist die Verfügbarkeit dieser Daten unerlässlich und ein Ausfall kann enorme Kosten verursachen.

1.2 Beschreibung der Idee

Das Ziel der Diplomarbeit ist es, eine massiv verteilte dokumentenbasierte NoSQL Datenbank zu entwickeln. Die Datenbank soll aus vielen einzelnen Nodes bestehen, die über ein API kommunizieren und so Ausfall und Fehler von einzelnen Nodes gemeinsam kompensieren. Für den Benutzer wird ein Portal zur Verfügung gestellt, welches eine einfache Benutzung der Datenbank erlaubt und die dahinter liegende Struktur transparent macht.
Das Portal stellt neben einer interaktiven Schnittstelle, ein Interface auf Basis von REST bereit, das von Clients verwendet werden kann.

Daraus ergeben sich die zwei Teile mit dem sich unser Projekt befasst:

- verteilte Datenbank
- Interface

Projektziele

1.3 MUSS Ziele

1.3.1 Entwickeln der verteilten Datenbank

Es soll eine massiv verteilte NoSQL Datenbank entwickelt werden, die die Daten als Dokumente abspeichert. Alle Datensätze sollen von allen Datenbank-Nodes parallel gespeichert und synchronisiert werden.

1.3.2 Kommunikation zwischen Datenbank-Nodes

Es soll ein Protokoll für die Kommunikation zwischen den Datenbank-Nodes zur Verfügung gestellt werden.

1.3.3 Entwickeln des User Interface

Es soll ein API auf Basis der REST-Architektur entwickelt werden, um unabhängig von der Programmiersprache mit den Nodes kommunizieren zu können.

1.3.4 C# Verwaltungssoftware

Es wird eine Verwaltungssoftware in C# entwickelt, mit der sich die Datenbank verwalten lässt.

1.3.5 Entwickeln einer Userauthentifizierung

Es wird eine User basierte Authentifizierung entwickelt.

1.4 Optionale Ziele (Soll, Kann Ziele)

1.4.1 Interaktiv - Entwickeln des User Interface

Das Interface wird zusätzlich noch als interaktive Shell implementiert.

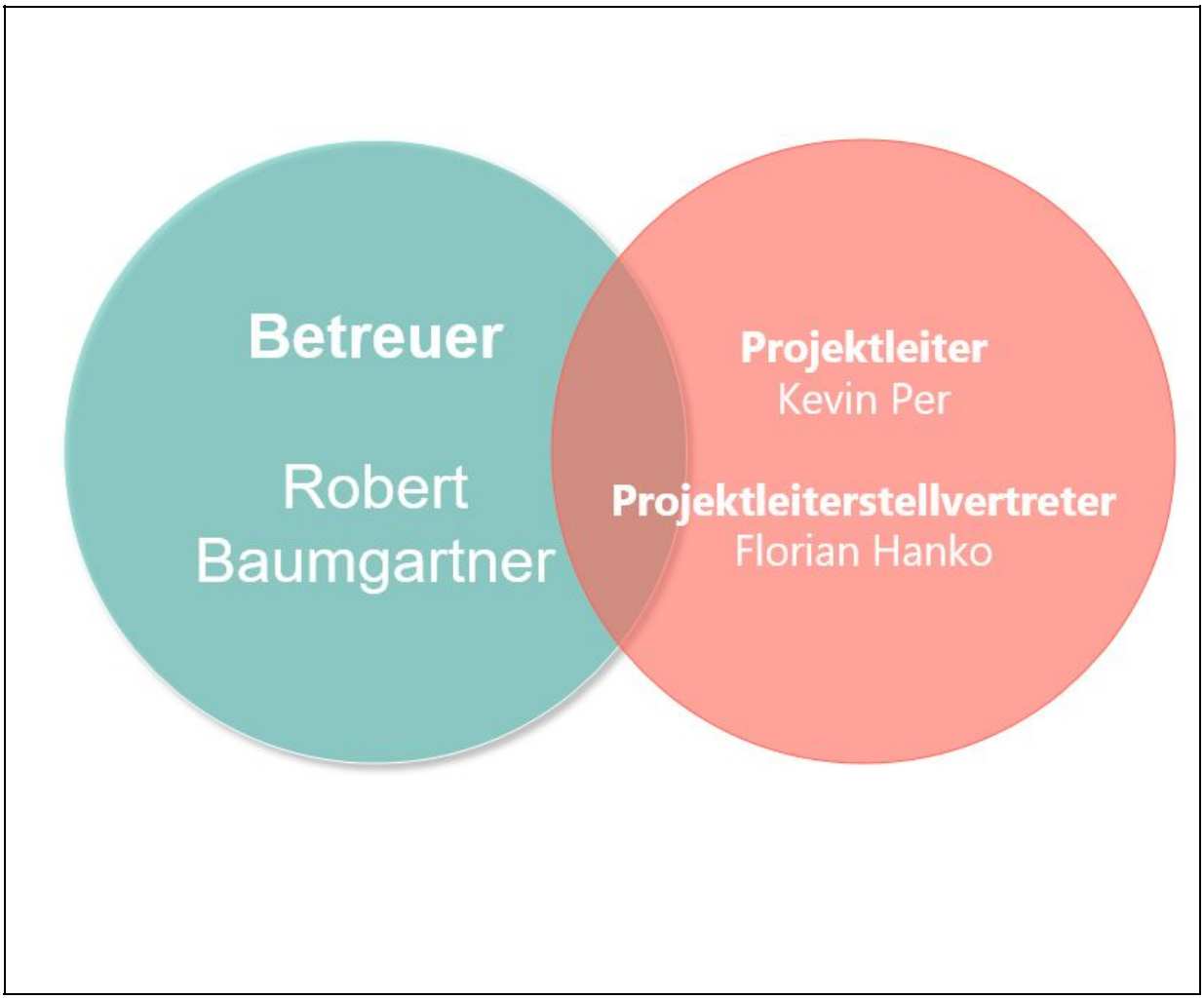
1.5 **NICHT Ziele**

1.5.1 Das Betreiben und Warten der Infrastruktur

1.5.2 Backup- und Restore-Funktionen für die Daten der Datenbank

2 Projektorganisation

2.1 Grafische Darstellung (Organigramm)



2.2 Projektteam

Funktion	Name	Kürzel	E-Mail
Betreuer	Robert Baumgartner	RB	robert.baumgartner@htl-ottakring.ac.at
Projektleiter und Projektmitarbeiter	Kevin Per	KP	k.per98@htl-ottakring.ac.at
Projektmitarbeiter	Florian Michael Hanko	FH	f.hanko98@htl-ottakring.ac.at

2.3 **Individuelle Aufgabenstellung**

2.3.1 Kevin, Per, Projektleiter

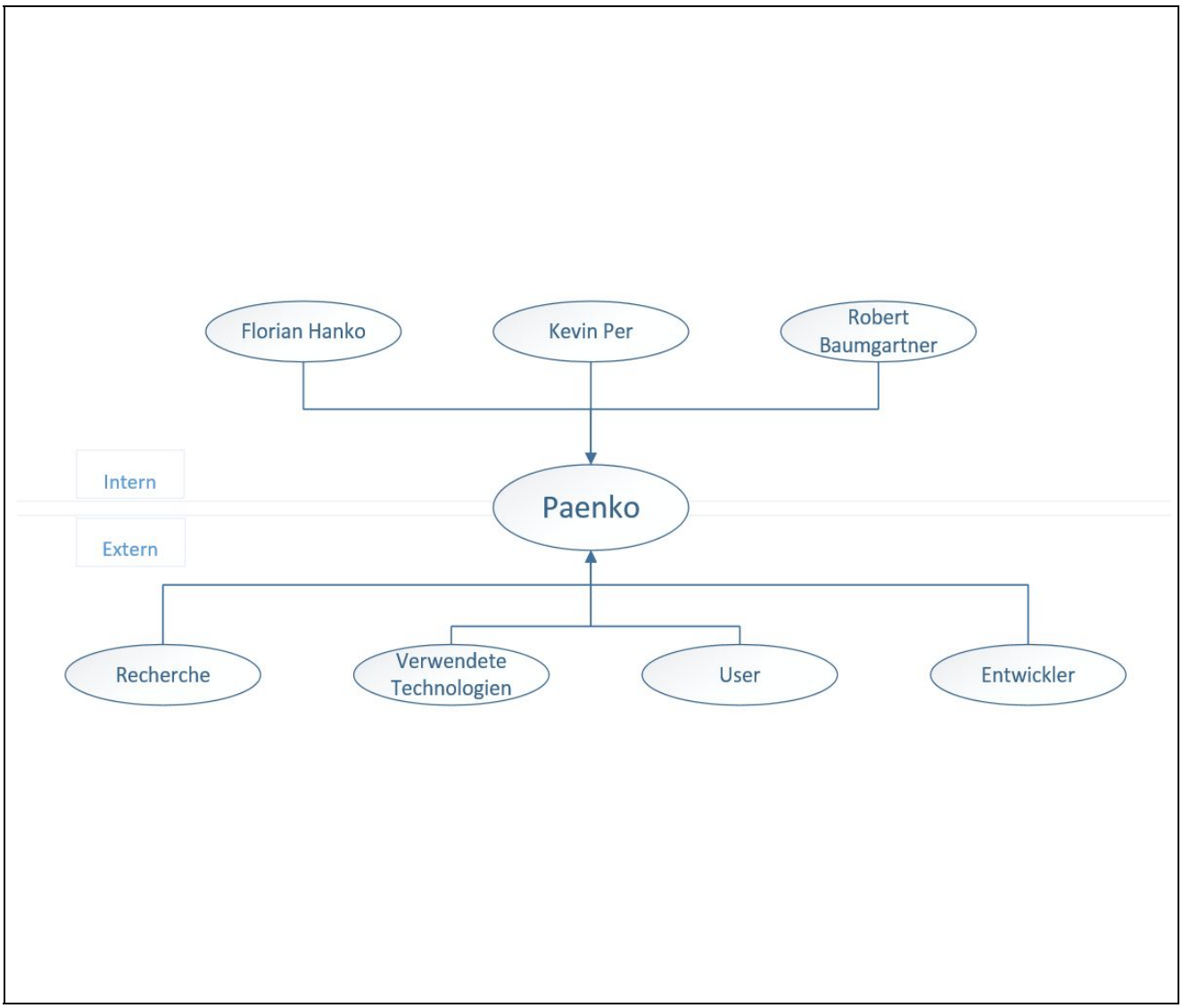
- Datenbank
  - Funktionen
    - Abfragen von Dokumenten
    - Update von Dokumenten
  - Testumgebung bauen
  - Test
    - Systemtest
    - Unit-Testing
  - REST API entwickeln
  - Userauthentifizierung
  - Webinterface entwickeln

2.3.2 Florian, Hanko, Stellvertretender Projektleiter

- Datenbank
  - Funktionen
    - Erstellen von Dokumenten
    - Löschen von Dokumenten
  - Dokumentation
- Interface
  - C# Sharp Library
    - Entwickeln
    - Dokumentation
  - C# Verwaltungssoftware

3 Projektumweltanalyse

3.1 Grafische Darstellung



3.2 Beschreibung der wichtigsten Umwelten

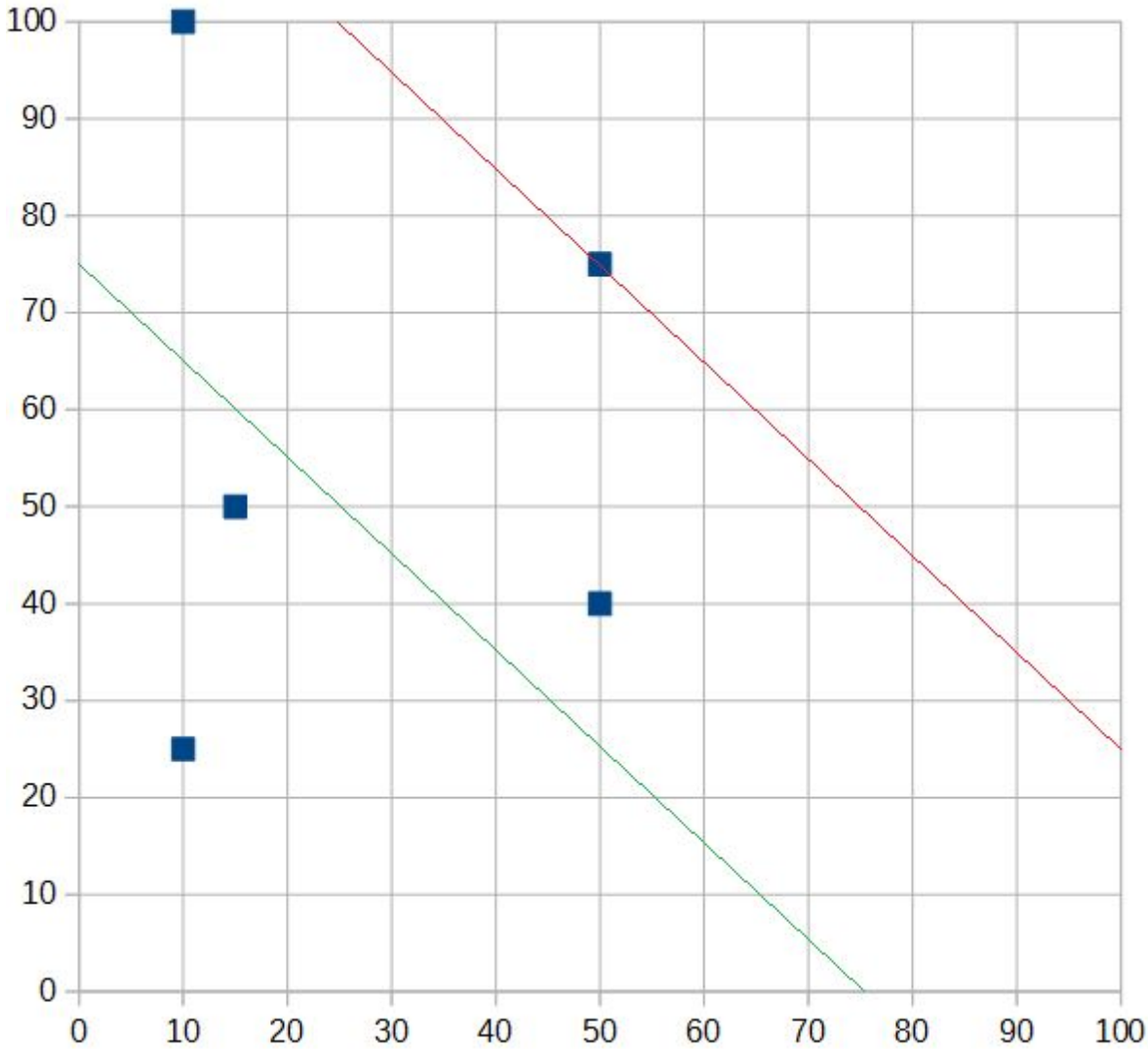
#	Bezeichnung	Beschreibung	Bewertung
1	Kevin Per	Projektleiter	+
2	Florian Michael Hanko	Projektleiterstellvertreter	+
3	Prof. Baumgartner	Kenntnisse in Datenbanksysteme und Software Entwicklung	+
4	Recherche	Sammeln von Informationen vor Beginn der Umsetzung	o
5	Verwendete Technologien	Technologien, die wir benutzen	+
6	User	Benutzer die die Datenbank verwenden	o
7	Entwickler / Open Source community	Personen die unser Projekt weiterentwickeln wollen	+

4. Risikoanalyse

3.3 Beschreibung der wichtigsten Risiken

#	Bezeichnung	Beschreibung des Risikos	P	A	RF
1	Unterschätzung	Zu viel Arbeit	50	75	3750
2	Technische Komplexität	Das Thema ist zu komplex	50	40	2000
3	Krankheit	Ausfall eines Teammitglieds	15	50	750
4	Datenverlust	Aufgrund von technischen Gründen Verlust des Projektes	10	100	1000
5	Technologien nicht mehr unterstützt	Ausgewählte Technologie nicht genügend unterstützt	10	25	250

3.4 **Risikoportfolio**



3.5 **Risiko Gegenmaßnahmen**

#	Bezeichnung	Gegenmaßnahme
1	zu viel Arbeit	sich auf den wichtigen Kern konzentrieren
2	technische Komplexität	Recherche / Vorbereitung
3	Krankheit	Risiko wird hingenommen
4	Datenverlust	Backups auf Github
5	Technologie nicht mehr unterstützt	Ähnliche Technologie verwenden

4 Meilensteinliste

Darstellung der Meilensteine mit geschätzten Terminen

Datum	Meilenstein
30.6.2016	Antrag fertiggestellt.
16.09.2016	Ansuchen der Diplomarbeit abgegeben
19.09.2016	Funktion implementiert, um Dokumente zu erstellen, die auf allen Nodes synchronisiert werden.
3.10.2016	Funktion implementiert, um Dokumente auf allen Nodes zu löschen.
28.10.2016	Website fertiggestellt
31.10.2016	Funktion implementiert, um Dokumente abzufragen
09.01.2017	Funktion implementiert, um Dokumente auf allen Nodes upzudaten.
09.01.2017	Ein REST API wurde implementiert.
9.01.2017	Userauthentifizierung implementiert.
23.01.2017	Eine Shell entwickelt, um das REST API um die Nodes anzusprechen.
06.02.2017	Eine C# Library entwickelt, um die REST-Schnittstelle zu abstrahieren.
06.03.2017	Erster Draft des Diplomarbeitsbuches fertig
06.04.2017	Diplomarbeitsbuch fertiggestellt

7 Kostenabschätzung

Abschätzung der Kosten des Projekts

#	Beschreibung der Kostenursache	Kosten
1	Github	0€
2	Slack	0€
3	Trello	0€
4	Google Apps	0€
5	Hosting der Nodes	100€
SUMME		100.00€

7.1 Finanzierung

Es wird versucht über Förderungen von Wettbewerben die Kosten des Projektes zu decken, andernfalls wird in der Informationstechnologie Abteilung der entsprechende Antrag gestellt.

8 Motivation

8.1 Kevin Per

Ich bin motiviert, weil ich davon Überzeugt bin, dass mir dieses Projekt dabei helfen wird, neue Erfahrung im Bereich der Software-Entwicklung zu erlangen.
Außerdem freue ich mich besonders NoSql kennenzulernen, weil ich mir überlege in Datenbanksystemen zu maturieren und ich mir erhoffe, Konzepte zu lernen, die mir im Unterricht helfen können.
Dieses Projekt wird mich herausfordern und ich freue mich auf die Arbeit mit meinem Kollegen Florian Hanko.

8.2 Florian Hanko

Seit ich Schüler an der HTL Ottakring bin beschäftige ich mich in meiner Freizeit mit Software Entwicklung. Ich arbeite gerne an schweren und herausfordernden Aufgaben, da das Thema dieser Dimplomarbeit sehr umfangreich und kompliziert ist, und auch weil ich im Schulgegenstand Datenbanksysteme maturieren möchte, finde ich, dass das Thema dieser Diplomarbeit perfekt für mich ist.