INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Softwareentwicklungspraktikum

Wegewahl in Abhängigkeit externer Datenquellen

Baumann, Oliver
Hoffmann, Werner
Popescu, Miriam
Schmidt, Dawin

Draft vom 23. Oktober 2013

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Softwareentwicklungspraktikum

Wegewahl in Abhängigkeit externer Datenquellen

Baumann, Oliver
Hoffmann, Werner
Popescu, Miriam
Schmidt, Dawin

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	**Dr. Nils gentschen Felde** LFE Kommunikationssysteme und Systemprogrammierung, LMU **Bruno Klauser** Consulting Engineer, Cisco Systems, Inc.
Abgabetermin:	28. Oktober 2013

Hiermit versichern wir, dass wir den vorliegenden Projektbericht selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 23. Oktober 2013

. . . . . .
(Baumann)

München, den 23. Oktober 2013

. . . . . .
(Hoffmann)

München, den 23. Oktober 2013

. . . . . .
(Popescu)

München, den 23. Oktober 2013

. . . . . .
(Schmidt)

Abstract

In dieser Projektarbeit sollte eruiert werden, ob es möglich ist, Daten, die keinen technischen (jedoch einen logischen) Zusammenhang mit einem Netz haben, für Routingentscheidungen zu verwenden. Hierzu wurde auf Basis einer Cisco API eine Software entwickelt, die es dem Nutzer erlaubt, einfacher auf Netzkomponenten zuzugreifen. Als Grundlage für Metrikentscheidungen werden hierbei Daten-Feeds genutzt, die aus unterschiedlichen Quellen bezogen werden können.

Inhaltsverzeichnis

1 Einleitung

Im Rahmen eines Softwareentwicklungs-Praktikums ausserhalb der Reihe an der Ludwig-Maximilians-Universität München ist eine quelloffene Software entstanden, die es dem Anwender ermöglicht, ein in seiner Kontrolle befindliches Netz anhand selbstgewählter Kriterien zu verwalten. Der vorliegende Bericht liefert eine einleitende Aufgabenbeschriebung, einen Architekturentwurf sowie eine Beschreibung der individuellen Arbeitspakete der vier Autoren und dokumentiert damit die Leistungen und Ergebnisse des Praktikums.

1.1 Motivation

Zur Entstehung des Internet und allgemein von Computernetzen war es von Bedeutung, dass Nachrichten zuverlässig ankommen. Bereits mit der Einführung des ARPANET machte man sich Gedanken zu „Network performance goals“.

Heutzutage sind die verlässliche Zustellung sowie ein angemessener *Quality of Service* sichergestellt und man kann auf zusätzliche Aspekte eingehen. Zur Wegewahl ziehen Routing-Protokolle vor allem intrinsische Metriken heran, welche Gegebenheiten wie etwa Bandbreite, Jitter oder Hop-Count abbilden. Implementierungen für externe Metriken existieren bisher in keinem gebräuchlichen Umfang. Dennoch können Situationen aufkommen, in denen äußere Kriterien für die Routing-Entscheidung herangezogen werden müssen, etwa wenn monetäre Kosten einer Verbindung von Bedeutung sind. Zwar existieren Verfahren wie Policy-Based Routing, bei dem anhand von Paketgröße, Quelladresse oder anderen Informationen im Paketheader Entscheidungen getroffen werden können; netz-externe Werte werden aber auch hier oft nicht oder nur statisch berücksichtigt.

Offenbar ist es nur schwer möglich, benutzerspezifische Metriken zu erfassen und abzubilden. Dem Benutzer bleibt lediglich der Weg der manuellen Konfiguration einzelner Knoten (z.B. bei OSPF das Einstellen der Bandbreite eines Links), was aber bereits in einem kleinen Netz unnötigen Verwaltungsaufwand erzeugt. Eine intuitiv zu bedienende, leicht zu wartende und zentrale Verwaltungsinstanz, die trotzdem die Autonomie des Netzes weitgehend erhält, existiert in quelloffener Form nicht.

1.2 Aufgabenbeschreibung

Ziel dieses Entwicklungsprojekts ist eine frei verfügbare Software, die es dem Nutzer ermöglicht, eigene Metriken zu definieren und Kontrolle über die Verbindungskosten und damit die Wegewahl im Netz zu erlangen. Der Anwender erhält die Möglichkeit, das Netz entsprechend spezifischer Vorgaben wie den bereits genannten monetären Kosten oder auch ökologischen Richtlinien zu konfigurieren. Hierzu sollen verschiedene Metriken angelegt und auf das Netz angewendet werden können. Zudem soll die Möglichkeit bestehen, auf externe Datenquellen

(*Feeds*) zuzugreifen und diese zur Kostenberechnung heranziehen zu können. Die Software leistet damit den Schritt weg von intrinsischen, hin zu externen und veränderbaren Metriken.

Der Nutzer greift auf die Applikation über eine Web-Oberfläche zu, die die Erstellung von Metriken, den Zugriff auf externe Datenquellen sowie die Verwaltung des Netzes ermöglicht. Die erwähnten externen Datenquellen sollen in einer Datenbank gespeichert werden, ebenso die nutzerdefinierten Metriken. Um auf das Netz zugreifen zu können ist es nötig, eine Schnittstelle zwischen der Software und dem zu verwaltenden Netz auszubilden. Über diese Schnittstelle werden einerseits Informationen über die Topologie des Netzes sowie einzelne Knoten an die Software, andererseits Konfigurationen an das Netz übertragen.

Die Aufgabenstellung umfasst jedoch nicht nur die Programmierung der Anwendung, sondern auch die Erarbeitung der Idee an sich samt Recherche zu bestehenden Möglichkeiten und Lösungsansätzen, die zeitliche Planung des Arbeitsablaufs, die Wahl einer geeigneten Entwicklungsumgebung samt nötiger Werkzeuge (Versionsverwaltung, Testumgebung) sowie die persönliche Organisation der Studenten untereinander (Kommunikationswege, Dokumentablagen). Ferner sind auch die vorliegende Dokumentation sowie ein Vortrag vor dem Kollegium der Lehr- und Forschungseinheit für Kommunikationssysteme und Systemprogrammierung Bestandteil der Arbeit.

2 Projektüberblick und -design

2.1 Projektüberblick

Im vorhergehenden Abschnitt wurden bereits einige Anforderungen an die Applikation genannt, die hier noch einmal aufgeführt werden sollen:

- Verwaltung und Überwachung des Netzes
- Web-Oberfläche als Nutzerschnittstelle
- Anbindung externer Datenquellen
- Datenbankanbindung

Zudem ergibt sich eine weitere Anforderung: eine Kontrollinstanz, die Schnittstellen zu allen anderen Anforderungen ausbildet und die Kommunikation der einzelnen Module untereinander realisiert.

Bevor diese Anforderungen in Module gefasst und je einem Entwickler zugeteilt wurden, erfolgte Recherche über nutzbare Technologien und bestehende Ansätze.
Auf Grund bestehender Kontakte zu **Cisco Systems, Inc.** ergab sich im Rahmen der Nachforschung über Ansätze zur Verwaltung von Netzen die Möglichkeit der Nutzung des *One Platform Kit* (onePK). Diese API ermöglicht die Kommunikation einer Software mit Cisco-Komponenten und steht für die Sprachen C und Java zur Verfügung. Wegen bestehender Erfahrung aller vier Studenten im Umgang damit wurde schließlich Java als Projektsprache gewählt.

Da Java keine typische Sprache zur Entwicklung von Web-Applikationen ist, musste nun eine Lösung gefunden werden, die grafische Nutzerschnittstelle an den Applikationskern anzubinden. Dank der Unterstützung durch die Lehr- und Forschungseinheit für Programmier- und Modellierungssprachen der LMU konnte hierzu das **Play-Framework**[1] herangezogen werden. Es implementiert das MVC-Pattern und ist vollständig RESTful, es können also einzelne Java-Methoden auf HTTP-URLs abgebildet werden. Zudem benötigt das Framework keinen Java EE Applikations-Server, da es bereits einen Netty-Server mitbringt und „in sich selbst“ lauffähig ist[2]. Die flache Lernkurve und das einfache Aufsetzen einer lauffähigen Umgebung waren die Kriterien für die Entscheidung zugunsten Plays.

Dynamische Daten sollten aus externen Quellen bezogen werden, um bspw. einen Wechselkurs oder Werte eines Sensors zur Link-Kostenberechnung nutzbar zu machen. Damit kann die Wegewahl im Netz unmittelbar an die Umwelt gekoppelt werden um etwa in Abhängigkeit des Stromverbrauchs einen Link vor einem anderen zu bevorzugen. Die Datenquellen stellen die benötigten Informationen in einem maschinenlesbaren Format zur Verfügung und die

[1] `http://www.playframework.com`
[2] Weitere Merkmale und Technologien sind bitte der Website des Play-Framework zu entnehmen.

Applikation stellt in regelmäßigen Abständen Anfragen, um aktuelle Werte nutzen zu können.

Die Datenbank soll Konfigurationen speichern und bereit halten, sodass der Nutzer auf bestehende Daten zurückgreifen kann. Zu den Daten, die dabei gespeichert werden, zählen Metriken, Datenquellen und Informationen des Netzes (genauere Beschreibung siehe Abschnitt 2.2).

Die Kontrollinstanz schließlich übernimmt die Anbindung aller Module und stellt Kommunikationsbeziehungen her. Hier findet zudem die Berechnung der Link-Kosten statt, wozu Daten aus dem Netz und der Datenbank abgefragt werden.

2.2 Entwurf

Dem Anforderungskatalog folgend wurde das Gesamtprojekt in vier Module gegliedert und je einem Entwickler zugeordnet. Hierbei waren Interesse und Erfahrung ausschlaggebend, und es wurde darauf geachtet, dass die Module möglichst in sich geschlossen arbeiten. So hätte der Ausfall eines Teilbereichs nicht den Ablauf des Projekts gefährdet. Folgende Arbeitspakete ergaben sich:

- AP1: Controller-Logik (Werner Hoffmann) (Kapitel 3)
- AP2: Topology Discovery & Netzagenten (Dawin Schmidt) (Kapitel 4)
- AP3: Externe Datenquellen & Datenspeicherung (Oliver Baumann) (Kapitel 5)
- AP4: Web-basierte Nutzerschnittstelle (Miriam Popescu) (Kapitel 6)

2.2.1 Funktionale Beschreibung der Arbeitspakete

Arbeitspaket 1 übernimmt die Funktion der Kontrollinstanz. Im **Controller** treffen die Anfragen ein, die der Nutzer über die Web-Oberfläche stellt, und werden verarbeitet. Folgende Typen von Aktionen werden dabei unterschieden:

a) Auslesen der Netztopologie
b) Speichern von Daten
c) Lesen von Daten
d) Berechnung von Routen und deren Konfiguration auf den Endgeräten
e) Verfolgung des Pfads zwischen Quelle und Senke (*traceroute*)

Ein Auslesen der Topologie wird an AP2 weitergeleitet („Topology Discovery & Netzagenten“ in Abbildung 2.1). Das Ergebnis dieser Aktion ist ein Graph, der schließlich auf der Nutzeroberfläche angezeigt wird.
Speichern bzw. Lesen von Daten werden an AP3 weitergeleitet, welches die betreffenden Aktionen auf der Datenbank ausführt. Beim Speichern erfolgt eine Rückmeldung über den Ausgang der Operation, beim Lesen erfolgt wiederum Anzeige auf der Benutzeroberfläche.
Die Berechnung der Routen schließlich wird direkt in AP1 ausgeführt (näheres hierzu siehe Abschnitt 3.3.3). Zudem erfolgt in diesem Arbeitspaket die zeitgesteuerte Aktualisierung aller Daten. Dazu werden nach Ablauf einer Zeitspanne die externen Datenquellen abgefragt,

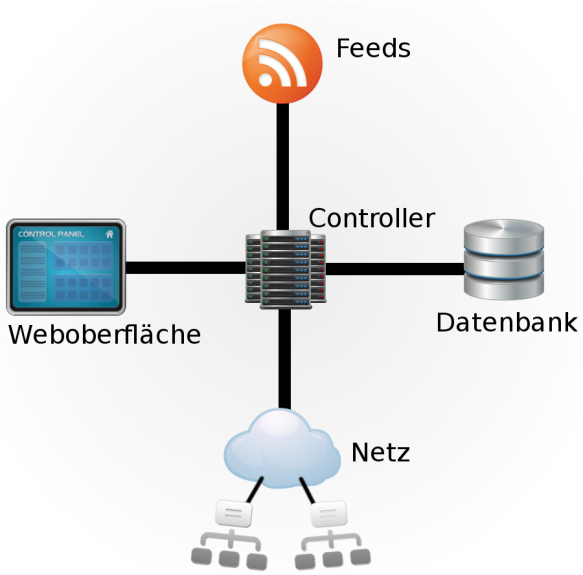


Abbildung 2.1: Modularisierung des Entwurfs

deren Werte als Link-Kosten gesetzt und schließlich die Routen berechnet und auf den Endgeräten konfiguriert.
Schließlich hat der Anwender die Möglichkeit, die aktuelle Konfiguration zu prüfen, indem ein traceroute ausgeführt wird. Hierzu werden Quelle und Senke über die Web-Oberfläche spezifiziert und der gewählte Pfad berechnet. Das Ergebnis wird dem Nutzer wiederum grafisch präsentiert.

Arbeitspaket 2 realisiert die Überwachung des zugrunde liegenden Netzes und die Konfiguration der individuellen Router. Wird eine Anfrage zum Auslesen der Topologie gestellt, liefert der **Topology-Discoverer** die nötigen Daten, die im Controller aus AP1 konsolidiert werden. Liefert der Controller Konfigurationen, werden diese an die **Netzagenten** verteilt und auf den Geräten ausgeführt. Diese Agenten können ferner Auskunft über den Zustand eines Knotens geben und etwa mitteilen, wenn ein Gerät ausgefallen ist.

In Arbeitspaket 3 werden zwei funktional unabhängige Komponenten implementiert. Zum einen findet hier die **Speicherung der Daten** statt, zum anderen werden hier **Daten aus externen Quellen** bezogen und nutzbar gemacht. Die Anbindung an ein Datenbank-Management-System ermöglicht das Speichern von Eingaben und deren Auslesen durch den Controller beim Programmstart, sodass dem Anwender eine konsistente Oberfläche zur Verfügung steht. Zudem werden in der Datenbank Netzkonfigurationen abgelegt (näheres hierzu in Abschnitt 5.4.3).
Die externen Datenquellen und deren Werte werden ebenfalls gespeichert und in diesem Modul für die Anwendung aufbereitet. Dazu ist es nötig, die Quelle der Daten zu spezifizieren und eine Auswertungsvorschrift anzugeben, anhand der ein einzelner numerischer Wert extrahiert werden kann. Diese Daten werden über die Benutzeroberfläche an den Controller und von hier an AP3 weitergereicht.

Arbeitspaket 4 umfasst die Erstellung der **web-basierten Nutzerschnittstelle**. Hier werden die Netz-Topologie anhand einer Karte visualisiert und Formulare zum Anlegen von Konfigurationen bereitgestellt. Die Kommunikation mit dem Applikationskern erfolgt, wie bereits erwähnt, über HTTP-URLs, Daten werden mit dem Controller bidirektional in einem maschinenlesbaren Format ausgetauscht.
Auf der Karte werden Knoten und Kanten dargestellt, mit denen interagiert werden kann; Konfigurationen können durch Anwählen einer Komponente eingesehen und verändert werden. Hier erfolgt zudem die grafische Repräsentation des traceroute-Vorgangs aus AP1.

2.2.2 Funktionale Beschreibung der Schnittstellen

Anhand Abbildung 2.1 sollen im Folgenden die Schnittstellen der Module funktional beschrieben werden. Die Abbildung lässt erkennen, dass jedes Modul (ausgenommen AP1, *Controller-Logik*) lediglich eine Schnittstelle ausbildet, nämlich zum Controller. Schnittstellen der Module untereinander existieren nicht, jede Kommunikation läuft über die zentrale Kontrollinstanz. Bei der folgenden Beschreibung soll dem Aufbau dieses Berichts gefolgt, also folgende Schnittstellen beschrieben werden:

1. Netz - Controller
2. Datenbank - Controller
3. externe Datenquellen - Controller
4. Web-Oberfläche - Controller

Netz – Controller

Das Netz-Modul übermittelt die Topologie an den Controller. Dazu werden in einem ersten Schritt die Topologien aller Standorte eingelesen und in einer Graph-Struktur gespeichert. Dies leisten die *Satelliten*, die je für ein LAN zuständig sind. Die einzelnen Graphen werden von den Satelliten an den Controller übermittelt und in einem weiteren Schritt zu einer globalen Topologie zusammengefügt, die alle untergeordneten Netze enthält. Weiterhin treffen an dieser Schnittstelle Events ein, die die *Netz-Agenten* senden. Hierbei kann es sich etwa um Ausfälle von Komponenten handeln. Diese Ereignisse werden anschließend vom Controller entsprechend verarbeitet.
Zum Netz hin gerichtet werden nach entsprechender Berechnung Routing-Konfigurationen übertragen. Genauer wird dieser Vorgang in Abschnitt 3.3.3 beschrieben, weshalb hier nicht weiter darauf eingegangen wird.

Datenbank – Controller

Das Datenbank-Modul stellt dem Controller Methoden zum Lesen und Schreiben aus der resp. in die Datenbank zur Verfügung. Dabei werden *Feeds* (bislang als „externe Datenquellen“ bezeichnet), *Metriken* sowie *Links* gespeichert (Näheres in Abschnitt 5.4). Beim Lesen aus der Datenbank werden entsprechende primitive Datentypen an den Controller zurückgegeben, je nach angefragtem Wert. Beim Schreiben übermittelt der Controller wiederum einen primitiven Datentyp und die entsprechende Methode liefert einen boole'schen Wert zurück, je nach Ausgang der Operation.

Externe Datenquellen – Controller

Das Feed-Modul implementiert den Zugang zu externen Datenquellen. Alle Daten, die für den Zugriff auf eine externe Quelle nötig sind, werden in der Datenbank gespeichert, weshalb eine enge Beziehung zwischen diesem Modul und dem Datenbank-Modul besteht. Beim Anlegen eines Feeds übermittelt der Controller alle vom Nutzer spezifizierten Daten zuerst an die für die Feeds zuständige Klasse des Datenbank-Moduls zur Speicherung. Hier werden nun zuerst die Werte aus der angegebenen Quelle ermittelt und anschließend abgelegt. Der Controller hat nun die Möglichkeit, über einen entsprechenden Lese-Aufruf diese Werte abzufragen und zur Berechung der Link-Kosten zu nutzen. Hierbei werden, wie bereits im vorhergehenden Abschnitt erwähnt, primitive Datentypen, in diesem Fall ganzzahlig-positive Werte zurückgegeben.

Web-Oberfläche – Controller

Über die Web-Oberfläche interagiert der Nutzer mit der Software. Alle Benutzereingaben werden über HTML-Formulare an die Applikation übertragen, die mit Methoden des Play-Frameworks auf die Werte der Eingabefelder zugreifen kann. Die Übertragung an den Controller geschieht mittels der POST-Methode des HTTP-Protokolls.
Für Nutzeranfragen an die Software werden, wie bereits erwähnt, Java-Methoden auf URLs abgebildet. Somit kann die entsprechende Methode als Linkziel angegeben werden und eine GET-Anfrage an den Controller gestellt werden. Dieser übernimmt die Weiterleitung der Anfrage an die jeweiligen Module und liefert die benötigten Daten im JSON-Format an die Weboberfläche, wo mittels JavaScript die erhaltenen Daten weiter verarbeitet werden.

2.3 Projekt-Management

Wie bereits erwähnt war nicht nur die Programmierung der Applikation selbst Gegenstand des Projekts, sondern auch die Organisation des gesamten Ablaufs. Hierzu zählen die Zuweisung der Arbeitspakete an je einen Entwickler, die Kommunikation mit den Betreuern (LMU und Cisco Systems) sowie untereinander. Weiterhin musste auch eine produktive Entwicklungsumgebung geschaffen werden, samt Editoren, Zugängen zu Systemen, Versions- und Dokumentenverwaltung.

2.3.1 Team

Die Zuweisung der Arbeitspakete erfolgte Anhand Interessen und Fähigkeiten der vier Studenten.

AP Controller-Logik

Zuständiger Entwickler Werner Hoffmann
Modul-Beschreibung Kapitel 3
Beweggründe Interesse an Entwicklung der Verwaltungslogik der Software, sowie:

- Verständnis der nötigen mathematischen Grundlagen
- analytisches Denkvermögen zur Ausbildung der Schnittstellen

Weiterhin zuständig für den Großteil der Kommunikation mit den Betreuern der LMU

AP Topology Discovery & Netzagenten

Zuständiger Entwickler Dawin Schmidt
Modul-Beschreibung Kapitel 4
Beweggründe Interesse an Entwicklung der Logik zur Netzüberwachung und -verwaltung mit Unterstützung durch Cisco Systems, sowie:

- Verständnis der nötigen technischen Grundlagen von Computernetzen und Routing
- Erfahrung mit Cisco und der Materie wegen früherer Werkstudententätigkeit

Weiterhin zuständig für den Großteil der Kommunikation mit den Betreuern seitens Cisco Systems

AP Externe Datenquellen & Datenspeicherung

Zuständiger Entwickler Oliver Baumann
Modul-Beschreibung Kapitel 5
Beweggründe Interesse an Entwicklung der Datenbankanbindung und Erschließen externer Datenquellen, sowie:

- ausgeprägte Kenntnisse im Umgang mit MySQL
- Erfahrung mit Auszeichnungssprachen wie XML (Datenformat für Feeds)

Weiterhin zuständig für operationale Abläufe, Einbindung von Entwicklungswerkzeugen sowie technische Dokumentation

AP Web-basierte Nutzerschnittstelle

Zuständiger Entwickler Miriam Popescu
Modul-Beschreibung Kapitel 6
Beweggründe Interesse an Entwicklung der grafischen Nutzerschnittstelle, sowie:

- Interesse an Informationsvisualisierung
- Erfahrung im Umgang mit Web-Technologien (HTML, CSS, JavaScript)

2.3.2 Organisation der Abläufe

Die Kommunikation innerhalb des Teams fand auf verschiedenen Wegen statt. Zu Beginn des Projekts wurde eine Mailingliste eingerichtet, zu der alle vier Studenten Zugang hatten. Der Großteil aller Themen konnte damit über Emails geklärt werden. Außerdem fand auf diesem Weg die Kommunikation mit den Betreuern statt. Zusätzlich wurden Dienste wie Skype und Jitsi genutzt, um Themen und Probleme zu diskutieren, wenn ein Team-Mitglied nicht vor Ort anwesend war. Allerdings wurde in den meisten Fällen ein persönliches Treffen in den Räumen der LMU bevorzugt. Kommunikation mit Cisco Systems fand vorwiegend durch Nutzung deren WebEx-Infrastruktur[3] statt.

Die Programmierung fand in regelmäßigen gemeinsamen Sitzungen statt, die in einem Projekt-Raum der LFE abgehalten wurden. So konnten auftretende Probleme oder Verständnisschwierigkeiten direkt mit den zuständigen Entwicklern geklärt werden. In diesen Sitzungen wurden auch Dokumentation und Abschlussvortrag erarbeitet.

[3] `http://www.webex.de/`

2.3.3 Entwicklungsumgebung

Zur Entwicklung wurde die Eclipse-Umgebung verwendet, mit der jeder der vier Studenten zuvor bereits gearbeitet hatte. Software-Versionen wurden in Git verwaltet, wozu anfangs ein kostenloses, privates Repository auf der Plattform Bitbucket[4] eingerichtet wurde. Im weiteren Verlauf wurde das Repository auf einen dedizierten Projektrechner umgezogen, auf dem auch die eigentliche Applikation lief. Zu diesem Rechner wurde den Entwicklern Zugang via SSH eingeräumt.

Ein Ausrollen aktueller Versionen erfolgte nur indirekt über die Aktualisierung in Git, eigenständige Releases gab es nicht.

Tests wurden in einer virtuellen Umgebung durchgeführt. Hierzu wurde in VirtualBox eine passende Instanz aufgesetzt, in der ein Rechnernetz simuliert werden konnte. Mittels *Host-only networking* konnte vom Host- auf das Gastsystem zugegriffen werden, sodass der auf dem Hostsystem laufende Versionsstand der Software produktiv getestet werden konnte. Näheres zur Test-Umgebung siehe bitte Abschnitt 4.4.

Zu Beginn des Projekts wurde außerdem das frei verfügbare Projektmanagement-System Redmine[5] genutzt, in dem Tickets angelegt und Entwicklern zugewiesen werden können und das grundlegende Zeitplanung ermöglicht (Fälligkeiten, geschätzte vs. gebrauchte Zeit). Allerdings ebbte die Nutzung bald wegen des erhöhten Aufwands und keiner nennenswerten Vorteile ab, bis sie schließlich ganz ausblieb. Absprachen und Zuweisung von Aufgaben erfolgten per Email oder persönlich.

2.3.4 Dokumentenverwaltung

Anfangs wurde zur gemeinsamen Ablage von Recherchematerial und Skizzen sowie zur Organisation von Planungstreffen eine Instanz der Wiki-Software DokuWiki[6] verwendet, die auf dem Projektrechner bereitgestellt wurde. Mit weiterem Fortschritt der Programmierung wurde jedoch auch dieses System zunehmend weniger genutzt und blieb lediglich als Archiv erhalten, aus dem Informationen für die Betreuer bezogen werden konnten. Stattdessen wurde zur Ablage beliebiger Materialien (Versionen der onePK-API, Bilder, Aufnahmen, Artikel, Berichte, ...) eine Dropbox-Freigabe genutzt, zu der jeder Entwickler Zugang hatte. Dies stellte sich vor allem beim Abfassen der initialen Aufgabenbeschreibung sowie dieses Berichts als sehr hilfreich heraus, da bei der anfänglichen Verwaltung der jeweiligen Versionen nicht immer klar war, wer auf welchem Versionszweig Änderungen vorgenommen hatte.

Eine technische Dokumentation zur Einrichtung der lokalen Entwicklungssysteme wurde von Oliver Baumann unter Benutzung des Sphinx Dokumentations-Systems[7] abgefasst. Sie war ebenfalls auf dem Projektrechner abgelegt und konnte jederzeit über den Webbrowser aufgerufen werden[8].

[4] `http://bitbucket.org/`
[5] `http://www.redmine.org/`
[6] `http://www.dokuwiki.org/`
[7] `http://sphinx-doc.org/`
[8] Diese Dokumentation wird auf der Webseite der LFE im Rahmen der Veröffentlicung des Projekts bereitgestellt.

3 AP1: Controller-Logik (von Werner Hoffmann)

In diesem Arbeitspaket geht es darum, die gewonnenen Informationen aus Nutzereingaben, Feeds und Netz zu verarbeiten. Wie in Abbildung 3.1 zu sehen ist, ist dies die Zentrale Komponente in der alle Informationen zusammen laufen. Zudem ist sie auch Verbindungsglied zwischen den einzelnen Komponenten. Wird aus der Weboberfläche in die Datenbank geschrieben werden in diesem Arbeitspaket die Daten so aufbereitet, wie die Datenbank sie speichern kann.

Abbildung 3.1: Darstellung des Logik Teils in der Anwendung

3.1 Teilgebiete

Logik Programmaufruf, Steuerung, Initialisierung
Service koordinierende Stelle
Formel Auswertung der Formel zu Berechnung der Linkkosten
Metrik Daten zu den Metriken
Kommunikation Steuerung des Kommunikationsfluss

Für jedes Teilgebiet wurde eine eigene Java Klasse angelegt um auch die Software angenehm zu strukturieren.

3.2 Logik

In der ursprünglichen Softwarebeschreibung war die *main class* als Hauptklasse gedacht, die den Programmablauf steuert. Aufgrund unserer Architekturentscheidung, das Webframework play! zu verwenden, ist eine Mainklasse überflüssig geworden. Die Aufgaben, der *main class* werden von der Klasse *Logic.java* übernommen.
Eine besondere Herausforderung war es in diesem Zusammenhang, dass play! nur auf statische Methoden zugreifen kann. Eine statische Methode kann jedoch nur mit statischen Instanzvariabelen arbeiten. Da wir ab einem gewissen Zeitpunkt des Programmablaufs sicher sein müssen, dass Komponenten bereits bestehen, ist es nötig gewesen, eine eigene Softwareschicht einzuführen. Diese Softwareschicht prüft ob bereits eine Instanz der Logik-Klasse vorhanden ist und diese ggf. erstellt. In dieser Schicht werden alle frameworkspezifischen Befehle ausgeführt und die Aufbereitung der Daten in das entsprechende Format vorgenommen. Die Initialisierung bestimmter Programmteile wird im Konstruktor der Logic-Klasse vorgenommen, sie ist die erste Klasse des Programms, die konsistent verfügbar ist (vgl. Singleton-Pattern).

3.3 Service

Der Service-Teil ist zuständig für die Kommunikation der Komponenten innerhalb des Programms. Er besteht im wesentlichen aus drei Klassen. Zum einen der Klasse *Communicator.java*, der verantwortlich ist für die Kommunikation zwischen Applikation und Web-Frontend. Zweitens der Klasse *Logic.java*, diese ist für alle organisierenden und koordinierenden Teile verantwortlich. Sie ist zudem Kernstück des Programms, da sie alle anderen Softwareteile zu einem verbindet. Und einer Klasse *IterativerThread.java*, die dafür sorgt, dass nach einem gewissen Zeitablauf das Netz neu ausgelesen wird, die Feeds abgerufen werden und die Routen gesetzt werden.

3.3.1 Service - Feed

Die Aufgabe dieses Bereiches ist es, eine Liste von allen Feeds zu pflegen, die die derzeit angelegten Feeds enthält, neue anzulegen und Links zuzuweisen. Bei Programmstart werden die bereits gespeicherten Feeds aus der Datenbank gelesen und in eine Liste geschrieben. Die Methode zum Aktualisieren der Werte der Feeds werden vor jeder Neuberechnung aufgerufen. Mit Hilfe der Werte, die die Feeds liefern, werden die Kosten für einen Link berechnet. Details zu den Feeds befinden sich in in Kapitel 5.5.

3.3.2 Service - Datenbank

Hier werden zum Programmstart alle vom Benutzer bereits getätigten Einstellungen abgerufen und in die entsprechenden Listen eingepflegt. Während des Programmlaufs werden Werte in die Datenbank geschrieben und gelesen. Näheres hierzu siehe Kapitel 5.4

3.3.3 Service - Netz

In der Klasse *Logic.java* wird der TopologyDiscoverer aufgerufen, dieser liefert ein Graphenobjekt zurück, das für die weitere Verarbeitung verwendet wird. Nach dem Berechnen der Routen werden diese in ihre Teilpfade zerlegt und an die Agenten weitergegeben, um die neu berechneten Routen zu setzen.
Zudem ist es möglich, dass aus dem Netz Events kommen. Diese werden von einem Event Listener abgefangen und entsprechend darauf reagiert. Das hier verwendete Graphenobjekt nutzt JGraphT[9] für die Modellierung des Graphenobjektes. Details zur Verarbeitung des Netzes finden sich in Kapitel 4.

3.3.4 Service - GUI

Der Service bietet der GUI einen Webserver an, dies wird duch das play!-Framework realisiert. Darin werden der GUI alle benötigten Daten aus der Datenbank bereitgestellt. An dieser Stelle werden Eingaben des Nutzers ermöglicht. Die Daten hierfür werden aus dem Arbeitsspeicher und der Datenbank ausgelesen. Siehe auch Kapitel **??**

3.3.5 Iterativer Thread

Hier wird dem Anwender ermöglicht, den Prozess doWork() nach einem bestimmten Zeitablauf iterativ wiederholen zu lassen. Da es aus Sicht der Entwickler nicht möglich war, hier einen sinnvollen Wert zu wählen, ist es möglich, in der „Properties-Datei “ hierfür einen Wert anzugeben, der dann ausgelesen wird. Der Wert sollte so gewählt sein, dass er in Zusammenhang mit den verwendeten Daten-Feeds Sinn ergibt. Liefert ein Feed alle 24 Stunden neue Werte sollte auch der iterative Thread diese alle 24 Stunden abholen. Werden die Daten minütlich aktualisiert und verändern sie sich im Tagesverlauf zyklisch, wäre eine Abfrage alle paar Stunden wohl geeigneter.

3.3.6 doWork-Methode

Da dies die zentrale Methode der Anwendung ist, wird sie hier extra beschrieben. In Grafik 3.2 kann man bildhaft sehen, wie sie arbeitet. In dieser Methode werden alle verfügbaren Daten (Benutzereingaben, gespeichert über Datenbank, Topologie, Werte aus den Feeds) verwendet, um daraus die vorher festgelegten Metrikwerte für jeden Link zu berechnen. Aus den Kosten für die einzelnen Links werden dann mittels eines Dijkstra von jedem Knoten zu jedem anderen Knoten die entsprechenden kürzesten Pfade gesetzt. Dieser Prozess ist sehr zeitintensiv, da mehrere Gerätewechsel (Applikationsrechner →Sattelitenrechner →Agentenrechner →Router), verbunden mit Technikwechseln nötig sind.

[9] http://jgrapht.org/

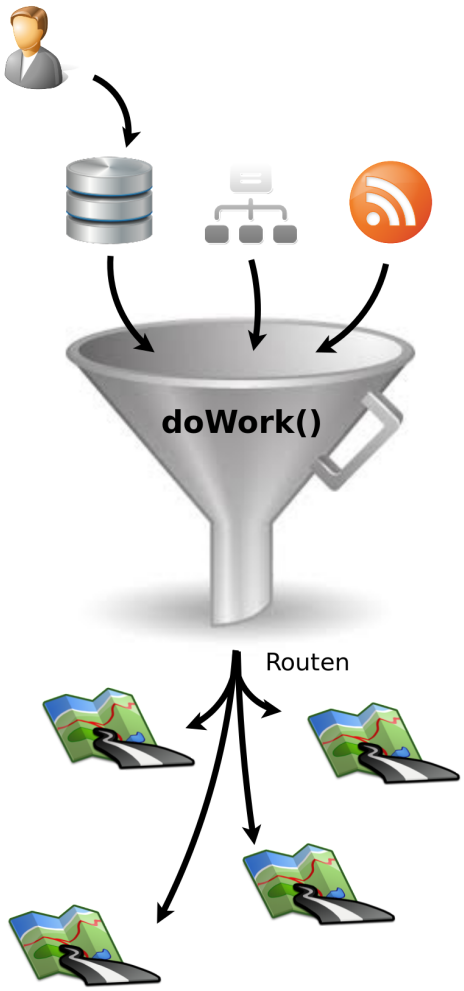


Abbildung 3.2: Darstellung der Funktionsweise von doWork

3.4 Formel

Mit unserem System ist es möglich, sich eine eigene Routingmetrik zu schreiben. Dies geschieht mit Hilfe einer Formel, die in der „Properties-Datei “ eingegeben wird. Die Formel kann neben den üblichen Rechenoperationen, feste Werte und Variabeln auch einige unäre Operationen wie sin, cos, floor oder ln beinhalten. Als Variablen sind die Namen der bereits angelegten Metriken zulässig. Bevor die Formel in einem Formelparser[10] geparst und dann berechnet wird, werden für jeden Link die Variablen (also die Metriknamen) durch die tatsächlichen Werte der Metrik ersetzt. Zum Ersetzen werden vorrangig die gespeicherten Feeds verwendet und ausgewertet. Ist für einen Link für diese Metrik kein Feed zugewiesen, wird der Standardwert der Metrik verwendet. Denkbar für Formeln wären u.a.: Routen mit niedriger Temperatur werden bevorzugt. Die Addition der Konstante 100 sorgt dafür, dass wir eine Division durch Null vermeiden und keine negativen Werte für Linkkosten berechnen. Dies könnte zu Schleifen im Dijkstra-Algorithmus führen.

```
1/(Temperatur+100)
```

z.B. aus Informationen einer Sicherheitsfirma

```
Gefahrenpotential
```

ein Feed gibt uns die Höhe der Produktion der auf unserem Dach befindlichen Solaranlage an, der zweite den aktuellen Verbrauch des Rechenzentrums. Der Standort mit dem größten Energieüberschuss wird bevorzugt.

```
Solarproduktion-Eigenverbrauch
```

3.5 Metrik

Unter Metrik verstehen wir hier nicht die mathematische Definition des metrischen Raumes. Der Begriff „Metrik“ wird bei uns verwendet für eine logische Schicht, die über dem gesamten Netz liegt. Jeder Link hat in jeder Metrik einen genau definierten Wert.
Zum Beispiel die Metrik „Hopcount“, bei der jeder Link des Netzes den Wert 1 hat. Eine Metrik definiert sich aus ihrem Namen, einer Einheit und einem Kostenwert für jeden Link. Um nicht zwingend für jeden Link einer Metrik einen Kostenwert angeben zu müssen, ist es erforderlich, einen Standardwert für die Metrik einzugeben. Der Zusammenhang zwischen Metrik, Feed und Link ist folgender:
Jede Metrik existiert für jeden Link.
Jeder Link erhält für jede Metrik maximal einen Feed.
Jeder Feed kann für mehrere Metriken und mehrere Links verwendet werden.
Dieser Zusammenhang soll durch Abbildung 3.3 noch einmal veranschaulicht werden.

[10] `http://kaiwitte.org/pr/jfunction/javadoc/`

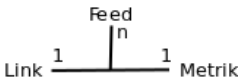


Abbildung 3.3: Darstellung der Beziehung zwischen Link, Metrik und Feed

Als Designannahme gilt, dass alle Metriken additiv sind, also keine multiplikativen oder konkaven Metriken existieren. Die beiden letztgenannten Arten wären durchaus denkbar (wenn auch nicht so häufig und komplexer zu bilden), jedoch unterstützt JGraphT diese beiden Operationen nicht.

3.6 Kommunikation

Die verschiedenen Möglichkeiten die *doWork* Methode zu starten sind in Abbildung 3.4 aufgezeit. Wir haben uns aufgrund der gegebenen Anforderungen dazu entschieden, keine klare Push- oder Pull-Kommunikation zu verwenden, sondern diese beiden Verfahren zu verwenden. Für die bereits oben angesprochene Methode *doWork* gibt es drei verschiedene Möglichkeiten, diese aufzurufen. Zum einen durch die Auswahl des Benutzers. Wenn der Benutzer wünscht, dass jetzt diese Methode ausgeführt wird, dann kann er das durch eine Schaltfäche auf der Oberfläche tun.
Zum anderen gibt es einen Thread, der kontinuierlich im Hintergrund läuft und immer nach einem bestimmten Zeitablauf den Prozess startet. Die Zeitdauer kann vom Benutzer gewählt werden. Sie ist so zu wählen, dass sie zu den Zeiträumen passt, in denen Veränderungen in den Feeds erwartet werden. Liegt ein Feed vor, der nur täglich aktualisiert wird, macht es kaum Sinn, ihn mehrmals täglich abzufragen. Dieser Thread wurde im Hinblick eingeführt, dass wir von einer externen Datenquelle keine Meldung (Events) bekommen können, wann eine Änderung stattgefunden hat. Diese Änderungen müssen nach regelmäßigen Zeitabständen von dieser Software überwacht werden.
Die dritte Möglichket, bei der die *doWork* Methode aufgerufen wird, ist durch den Agent. Erfährt dieser von Änderungen in seinem Teil des Netzes (z.B. Ausfall von Knoten, Kanten etc.) gibt er ein Netzevent aus, das dann wiederum zum Aufruf der *doWork* Methode führt.

Abbildung 3.4: Darstellung der verschiedenen Möglichkeiten den Hauptprozess zu starten

4 AP2: Topology Discovery & Netzagenten (von Dawin Schmidt)

Im Arbeitspaket „Controller-Logik“ werden einige Anforderungen definiert, welche für die Funktion der Anwendung obligatorisch sind (siehe Kapitel 3.2). Unter anderem werden verschiedene Operationen auf der Seite des Netzes, wie z.B. das Einlesen der Netz-Topologie oder auch das Konfigurieren von Routen, benötigt. Der folgende Abschnitt beschreibt den Lösungsansatz der gegebenen Problemstellung.

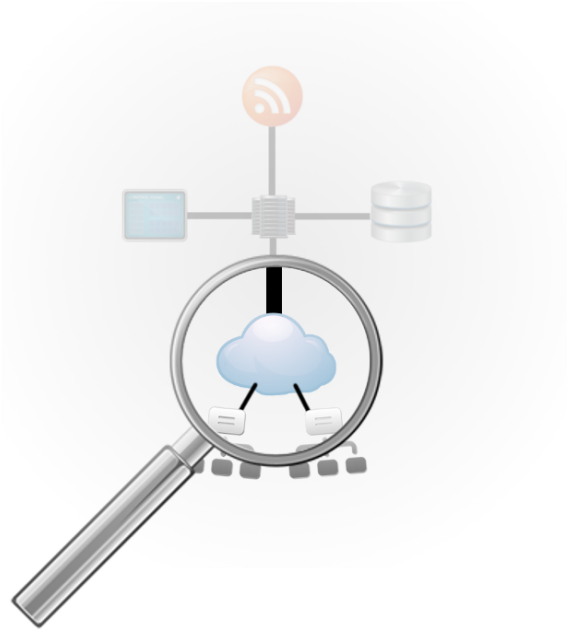


Abbildung 4.1: Arbeitspaket „Topology Discovery & Netzagenten“

4.1 Client-Server Architektur

Für das Bereitstellen von Management-Operationen auf Netz-Seite wurde eine klassische Client-Server Architektur gewählt, wobei der Client diese Funktionen zur Verfügung stellt. Im Rahmen dieses Projekts beinhaltet der Server die Arbeitspakete *Controller-Logik*, *externe Datenquellen & Datenspeicherung* sowie die *web-basierte Nutzerschnittstelle* und der Client die Teilbereiche *Satellit* und *Agent*. Aufgabe dieses Arbeitspakets war es zum einen, den Client zu implementieren und eine entsprechende Schnittstelle, den *Topology-Discoverer*, auf Server-Seite bereitzustellen. Der Client wurde in ein unabhängiges Software-Projekt ausgelagert, welches physisch (lauffähig auf anderen Hosts) und logisch (keine Abhängigkeiten zu

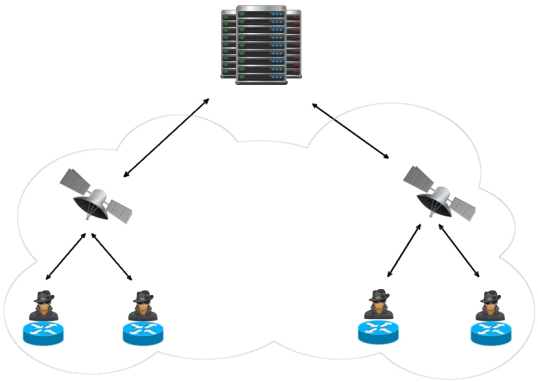


Abbildung 4.2: Server mit zwei Clients

anderen Arbeitspaketen) getrennt von der Server-Applikation ist.

Am Beispiel „Netz-Topologie einer Organisation einlesen“ wird der Client-Server Aufbau deutlich (siehe Abbildung 4.2): Eine Firma XYZ hat zwei Standorte, die in Deutschland verteilt sind. In jedem LAN der Standorte steht ein Host, auf dem der Client bestehend aus Satellit und Agent installiert ist. Agenten übermitteln die Topologie ihres Zuständigkeitsbereichs an den Satelliten. Der Satellit fügt die einzelnen Teilgraphen zusammen und übermittelt die Topologie des Standorts über Transitnetze an den Topology-Discoverer (Server). Dieser wiederum fügt die Graphen der Clients zusammen und ermittelt so die globale Topologie des Netzes.

Wie aus Abbildung 4.2 ersichtlich, besteht dieses Arbeitspaket aus mehreren Teilbereichen, auf welche im folgenden nun näher eingegangen wird:

Topology-Discoverer Schnittstelle zwischen Server-Applikation und Client
Client Software-Paket bestehend aus Satellit und Agent
- **Satellit** Server-Verbindungsmanagement und Agenten-Verwaltung
- **Agent** Administration einer Teilmenge eines Netzes
- **Deployment** Rollout des Clients

Test-Umgebung Virtuelle LAN-Infrastruktur

4.2 Topology-Discoverer

Der Topology-Discoverer (Discoverer) ist die Schnittstelle zwischen den einzelnen Clients und der Server-Applikation. Die Klasse verwaltet die Clients und ist physischer Bestandteil des Arbeitspakets *Controller-Logik*, jedoch logisch davon getrennt. Via Java-RMI kommuniziert der Discoverer mit den verteilten Clients, erteilt Kommandos (Methoden TopologyDiscoverer.getTopology() und TopologyDiscoverer.setRoute()) und reagiert auf Ereignisse (Methode TopologyDiscoverer.receiveEvent()). Zudem werden Topologie-Graphen, die von den Clients übermittelt werden, zusammengefügt und als einheitlicher Graph zur Verfügung gestellt.

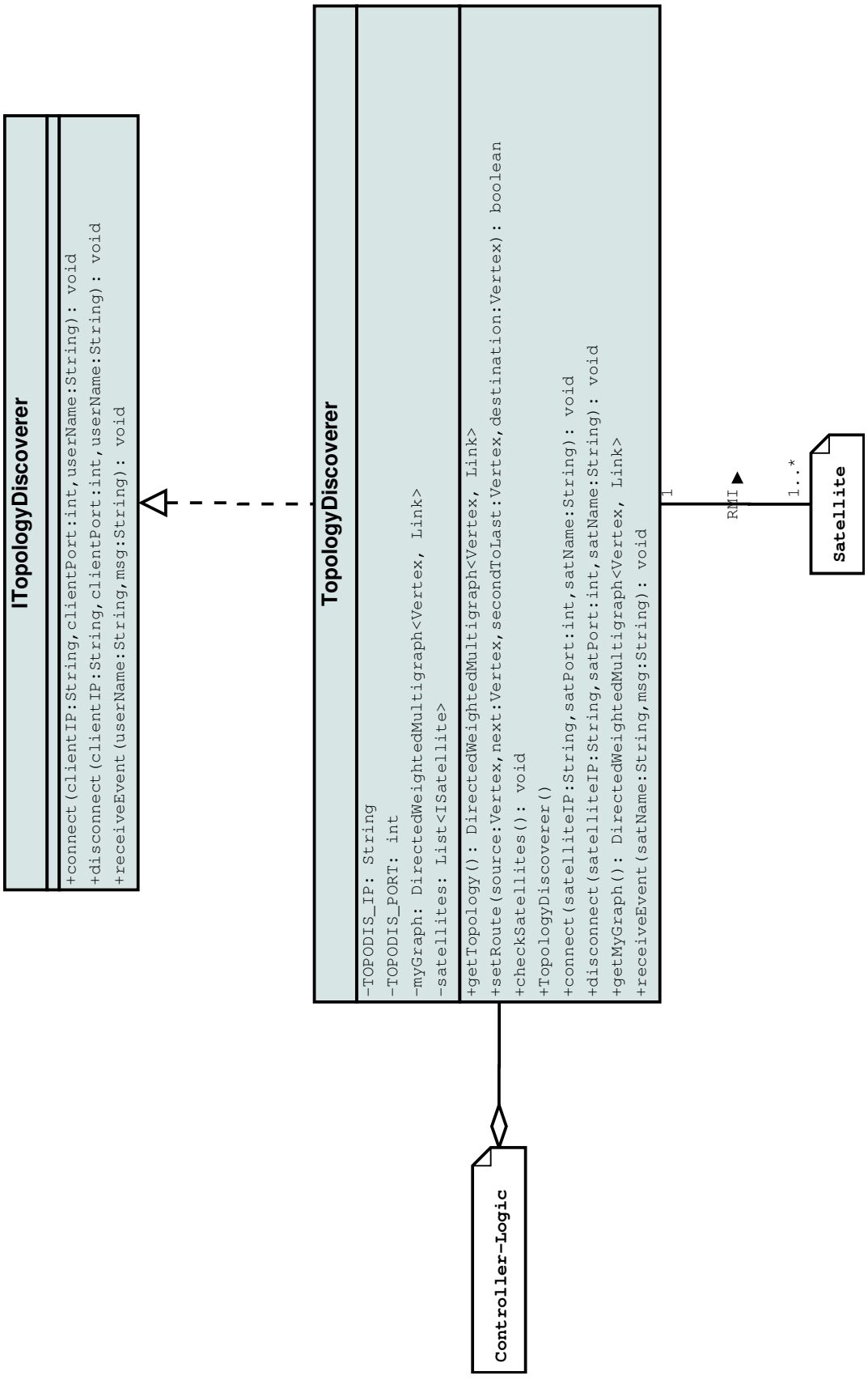


Abbildung 4.3: UML-Diagramm des Topology-Discoverer

Das UML-Diagramm aus Abbildung 4.3 zeigt die Beziehungen zwischen den verschienden Entitäten. Der Discoverer implementiert das Interface *ITopologyDiscoverer*, welches Methoden definiert, die via RMI remote zur Verfügung stehen. Desweiteren verwaltet der Discoverer einen oder mehrere Clients und ist Teil des Pakets Controller-Logik (Assoziations-Beziehung).

4.3 Client

Wie bereits erwähnt, besteht der Client aus zwei größeren Teilbereichen, dem Satelliten und dem Agenten. Auf beide Teile soll nun gesondert eingegangen werden.

4.3.1 Satellit

Der Satellit fungiert als Schnittstelle zwischen Discoverer und Agent indem dieser Befehle von beiden Seiten durchreicht. Eine Push/Pull-Kommunikation wird dadurch ermöglicht, dass der Satellit sowohl als RMI-Client als auch als RMI-Server implementiert ist. Typische Befehle können beispielsweise das Ermitteln der Topologie, das Konfigurieren von Routing oder die Mitteilung von Netz-Events sein. Des weiteren koordiniert der Satellit Befehle und muss sicher stellen, dass Anweisungen beim richtigen Agenten ankommen. Das Zusammenführen der Teilgraphen, welche von den Agenten übermittelt werden, ist eine weitere Aufgabe des Satelliten. Durch einen eindeutigen Identifier können doppelte Knoten bei diesem Schritt entfernt werden und als lokaler „Standort-Graph“ an der Discoverer übertragen werden. Weiterhin übernimmt diese Klasse das Verbindungsmanagement mit dem Discoverer: Beim Start der Applikation meldet sich der Satellit über einen RMI-Call bei dem Discoverer an, beim Beenden wird mithilfe eines Shutdown-Hooks die Abmeldung vom Discoverer ausgeführt.

Das UML-Diagramm aus Abbildung 4.4 zeigt die Modellierung des Satelliten: Die Klasse „Satellite“ erzeugt die Agenten-Objekte (hier beispielhaft aus der Klasse „CiscoAgent“) und steht via Java-RMI mit dem Discoverer in Verbindung. Entsprechende RMI-Server Funktionalität wird mit dem Interface „ISatellite“ angeboten. Die Klassen „Link“ und „Vertex“ modellieren jeweils eine Kante und einen Knoten und werden von der Bibliothek „JGraphT“ zur Verfügung gestellt.

4.3.2 Agent

Der Agent ist die ausführende Einheit und interagiert unmittelbar mit dem Netz. Dieser nimmt Kommandos des Satelliten entgegen und führt dieses aus, z.B. „lese Topologie“ oder „setze Route“. Über entsprechende Event-Listener und kontinuierliche Keep-Alive Tests kann der Agent Knoten-Ausfälle erkennen und darauf reagieren. Beispielsweise ruft der Agent bei Ausfall die „receiveEvent()“ Methode des Discoveres auf, welcher das Ereignis dann verarbeiten kann. Informationen beschaffen, Konfigurationen ändern und den Status überwachen sind schlussendlich die Hauptaufgaben des Agenten.

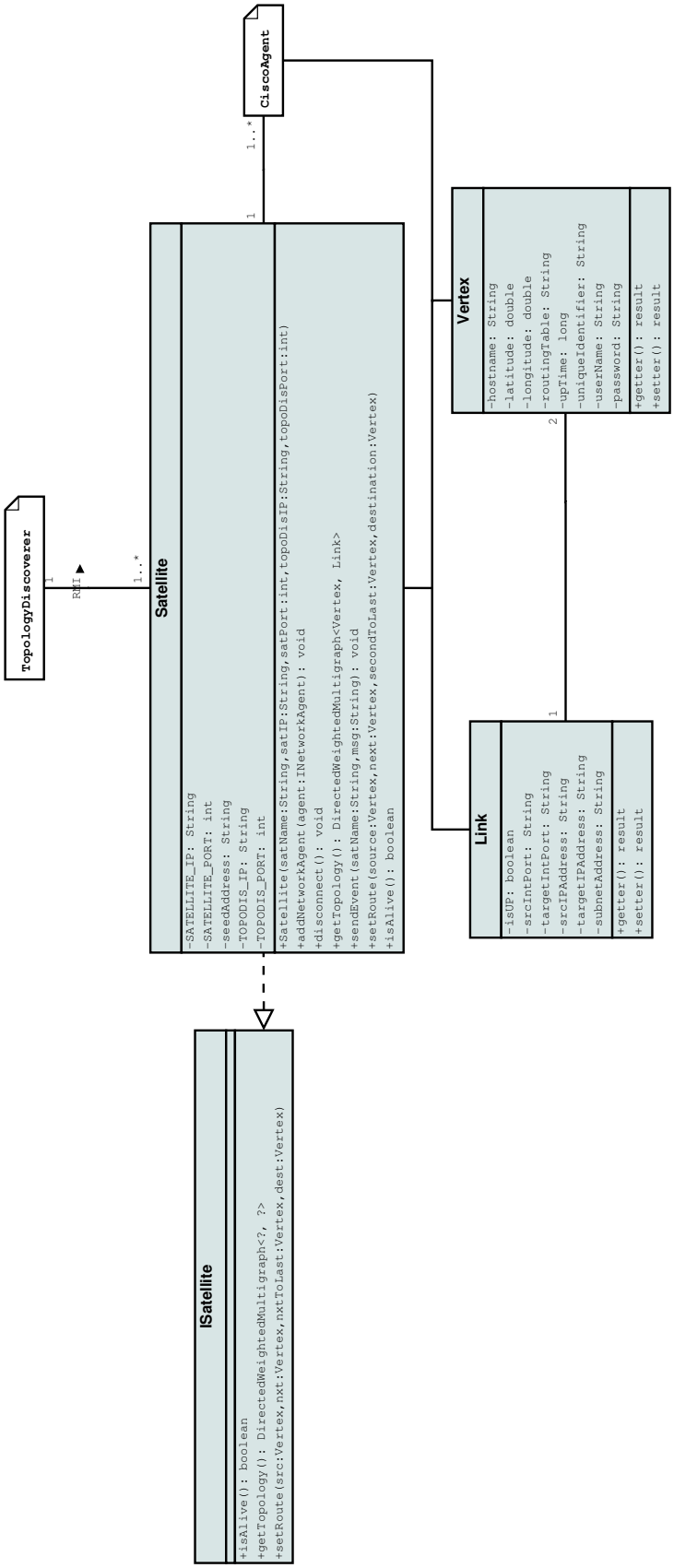


Abbildung 4.4: UML-Diagramm des Satelliten

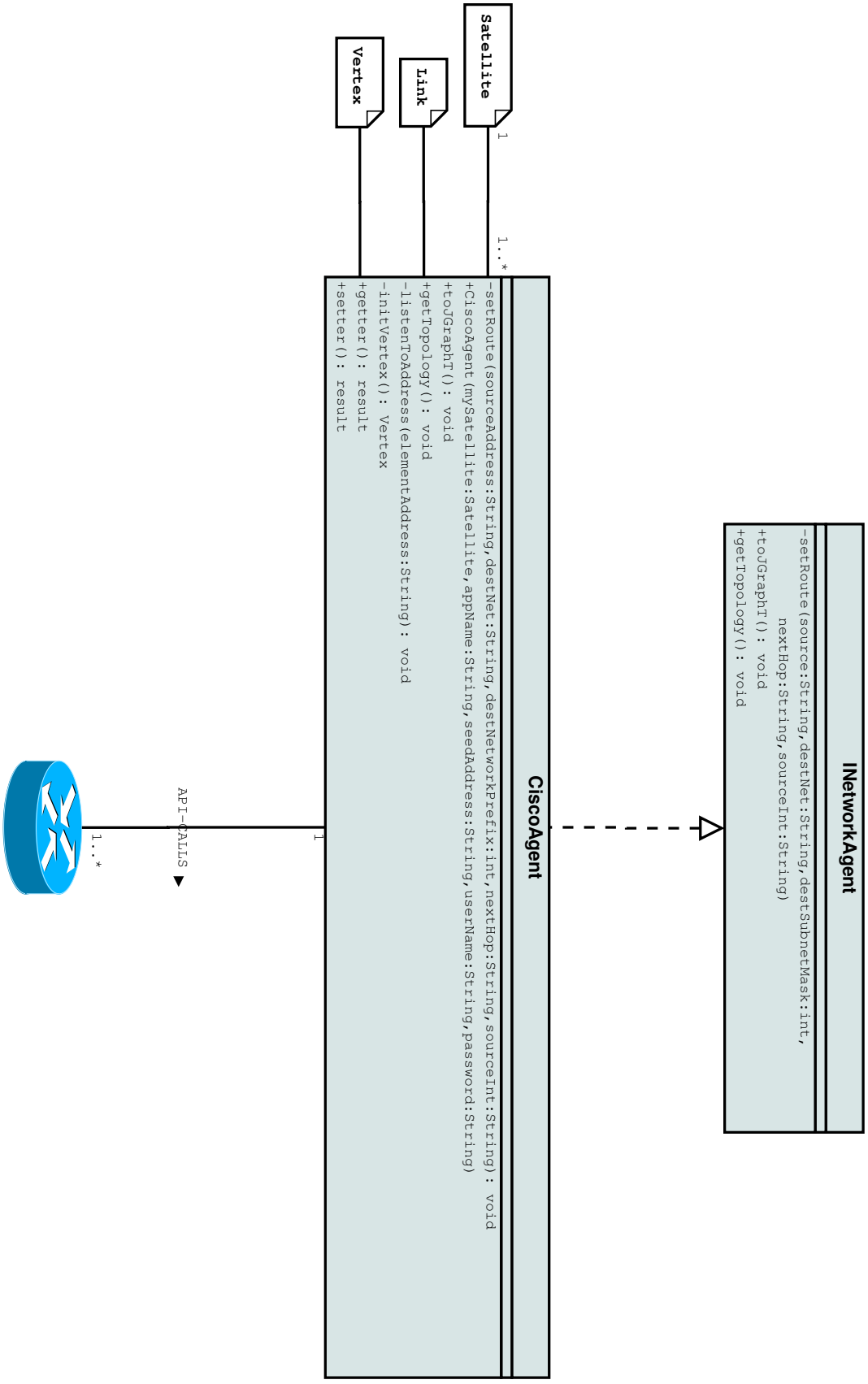


Abbildung 4.5: UML-Diagramm des Agenten

Am Beispiel von Cisco (Klasse „CiscoAgent“) kommuniziert der Agent mithilfe der Cisco eigenen API *onePK*[11] mit den Hardware-Komponenten. Damit diese Kommunikation funktioniert, muss eine entsprechende SDK-Version und eine spezielle IOS-Version verwendet werden. Theoretisch können auch SDKs/APIs anderer Hersteller verwendet werden - ein entsprechender Agent muss nur das Interface „INetworkAgent“ implementieren (siehe UML-Diagramm aus Abbildung 4.5).

4.3.3 Deployment

Für den Rollout eines oder mehrerer Clients werden folgende Design-Annahmen getroffen:

a) Benutzername und Passwort zur Authentifizierung zwischen Agent und Knoten sind auf jedem IOS-Gerät gleich.
b) Cisco Discovery Protocol ist auf jedem IOS-Gerät aktiviert (wird zur Topology-Discovery benötigt).
c) RMI-Port in Firewall freigeschaltet, Port kann gewählt werden.
d) GPS-Koordinaten der IOS-Geräte sind hart kodiert, da „Location Service Set“ Funktionalität im verwendeten onePK-SDK nicht verfügbar.
e) Es wurde nur auf virtueller Hardware getestet.

Eine Installations-Anleitung des Clients ist in englischer Sprache im Anhang verfügbar.

4.4 Test-Umgebung

Einen weiteren Teil dieses Arbeitspakets bildet die Einrichtung und die Administration einer Test-Umgebung. Auf einem Ubuntu-Server ist die Virtualisierungssoftware VirtualBox installiert, welche das Gastsystem Fedora 14 (Codename Laughlin) virtualisiert. In diesem Gastsystem wiederum wird ein virtuelles Netz mithilfe von *IOS-on-Linux* (IoL) simuliert (siehe verwendete Netz-Topologie als ASCII-Art in Listing 4.1). Außerdem bietet ein auf dem Gast installierter OpenVPN-Server einen Fernzugriff auf die VM an. Hierfür wurde eine Certificate Authority (CA) eingerichtet und Zertifikate an die Entwickler verteilt. Abbildung 4.6 zeigt den Versuchsaufbau: der Entwickler kann innerhalb des Münchner-Wissenschafts-Netzes mithilfe von OpenVPN auf die virtuelle Maschine zugreifen und verschiedene Simulationen im virtuellen Netz testen.

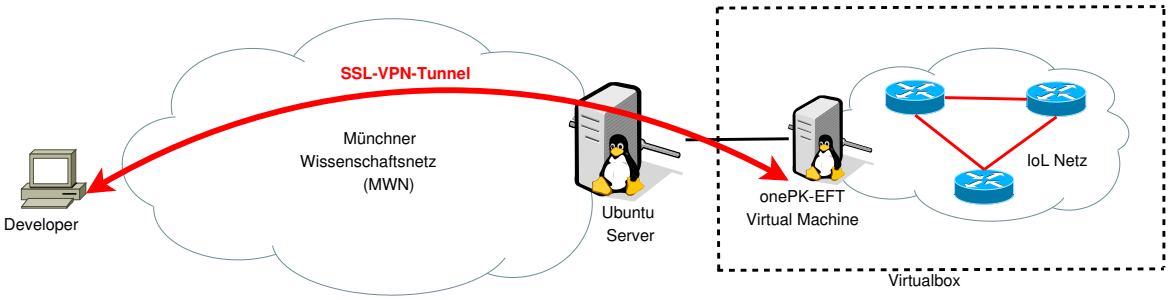
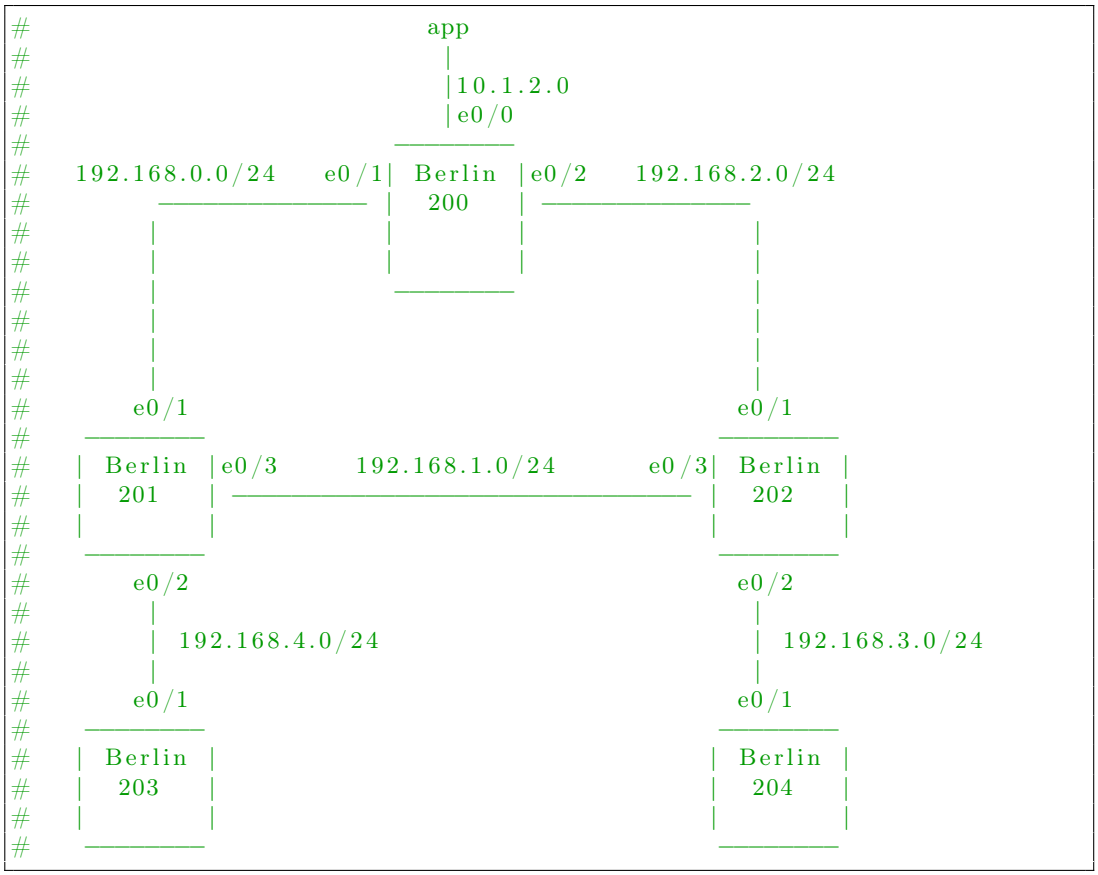


Abbildung 4.6: Aufbau der Test-Umgebung

[11] `http://www.cisco.com/en/US/prod/iosswrel/onepk.html`



Listing 4.1: Topologie des virtuellen IoL-Netzes

5 AP3: Externe Datenquellen & Datenspeicherung (von Oliver Baumann)

Wie einleitend bereits erwähnt ist es ein Ziel der Software, dem Nutzer eine Schnittstelle zu umweltbezogenen Daten zu liefern. Anhand dieser Daten erhält er die Möglichkeit, unmittelbar Einfluss auf Routingentscheidungen zu nehmen, indem er die externen Daten als Kostenwerte den Netzkanten zuweist. Dadurch ist es möglich, eine Wegewahl etwa in Abhängigkeit eines aktuellen Wechselkurses zu treffen, um Betriebskosten gering zu halten. Dazu muss der Anwender eine Quelle spezifizieren, aus der Informationen bezogen werden sollen (Feed), sowie eine Auswertungsvorschrift, anhand der sich die gewünschten Daten erschließen lassen. Diese Nutzereingaben werden dem Modul übergeben und anschließend verarbeitet. Das Resultat sind numerische Werte, die schließlich in der Datenbank gespeichert werden und zur Änderung der Link-Kosten genutzt werden können.

Neben den externen Quellen sind noch weitere Daten zu speichern; so hat der Nutzer etwa die Möglichkeit, Metriken anzulegen, die zusammen mit einem Feed einer Netzkante zugewiesen werden können. Sowohl Metriken als auch Kanten sind persistent zu speichern.

Abbildung 5.1: Feed- und Datenbank-Modul

5.1 Anforderungen

Folgende Anforderungen an das Arbeitspaket ergeben sich:

- bidirektionale Kommunikation mit Controller
- Speicherung und Auswertung von Daten-Feeds
- Speicherung von Metriken und Links
- Konfigurationen eines Datenbanksystems

5.2 Schnittstellen

Das Arbeitspaket bildet zwei Schnittstellen aus:

- Controller – Feeds
- Controller – Datenbank

Über die Schnittstelle zwischen Controller und Feed werden einerseits die Eingaben des Nutzers als primitive Datentypen übertragen. Das Feed-Modul wertet diese aus und speichert sie zusammen mit dem resultierenden Wert in der Datenbank, wo sie zur weiteren Verwendung verfügbar sind.
Über die Schnittstelle zwischen Controller und Datenbank laufen alle Lese- und Schreiboperationen. Stellt der Controller eine Lese-Anfrage, übermittelt er den passenden Primärschlüssel; das Datenbank-Modul gibt dann die zugehörigen Daten aus. Bei einer Schreib-Anfrage übermittelt der Controller die Eingaben des Nutzers und erhält als Rückmeldung einen boole'schen Datentyp, je nach Ergebnis der Operation.

5.3 Architektur

5.3.1 Datenbank-Modul

Das Datenbank-Modul stellt Lese- und Schreiboperationen zur Verfügung. Hierzu muss zuerst spezifiziert werden, welche Daten überhaupt in der Datenbank vorliegen: Relationen für Metriken, Feeds und Links müssen erstellt werden. Danach bietet es sich an, je nach Relation eigene Methoden zu definieren, auf die der Controller zum Lesen bzw. Schreiben zugreifen kann. Der Grund dafür liegt in der Funktion des Play-Frameworks: hier werden Java-Klassen über eine objektrelationale Abbildung als Relationen in einer Datenbank verfügbar gemacht. Für jede Relation existiert also eine individuelle Klasse, in die sich Lese- und Schreib-Methoden auslagern lassen. Der Controller kann dann je nach Typ auf die entsprechenden Funktionen zugreifen. Als Datenbanksystem wird in diesem Zusammenhang MySQL verwendet.

5.3.2 Feed-Modul

Das Feed-Modul erschließt externe Datenquellen für die Konfiguration des Netzes. Hierzu muss ein maschinenlesbares Datenformat implementiert werden, auf das der Nutzer online zugreifen kann. Aufgrund breiter Verfügbarkeit wird hier XML in Verbindung mit XPath genutzt. Die URL des XML-Dokuments sowie der XPath, der den Ort der Daten innerhalb des Dokuments angibt, müssen in der Datenbank gespeichert werden. Zuvor erfolgt der Abruf des XML-Feeds und die Auswertung des XPath-Ausdrucks gegen das Dokument, um einen einzelnen ganzzahlig positiven Wert zu erhalten. Der Controller kann dann mittels einer Lese-Anfrage auf diese Werte zugreifen.

5.4 Datenspeicherung – Implementierung

Wie bereits angesprochen wurde als Datenbanksystem MySQL gewählt. Der MySQL-Server läuft auf dem selben Projektrechner wie auch die Anwendung selbst, kann in der Konfiguration also als *localhost* spezifiziert werden. Das Play-Framework untertsützt MySQL standard-

mäßig mittels JDBC-Treiber, damit ist also bis auf die Angaben zum Verbindungsaufbau in **/conf/application.conf** keine weitere Konfiguration nötig (siehe Listing 5.1).

```
1  [...]
2
3  # Database configuration
4  # ~~~~~
5  # You can declare as many datasources as you want.
6  # By convention, the default datasource is named `default`
7  #
8  db.default.url="jdbc:mysql://127.0.0.1/play_dev"
9  db.default.driver=com.mysql.jdbc.Driver
10 db.default.user=USERNAME
11 db.default.password=PASSWORD
12
13 [...]
```

Listing 5.1: Datenbank-Verbindung in application.conf

Play nutzt Ebean[12] zur objektrelationalen Abbildung, weshalb sämtliche Relationen als Java-Klassen verfasst werden können. Diese Klassen werden beim Programmstart in Tabellen der Datenbank übersetzt, die Felder der Klassen entsprechen den Attributen der Tabellen; zusätzliche Hilfsmethoden, bspw. zum Suchen oder Speichern der Entitäten, können definiert werden. Listing 5.4 am Ende dieses Kapitels zeigt ein Modellierungsbeispiel anhand der Metrik-Relation.

Bei der Implementierung der Datenbankanbindung wurde ein Façade-Pattern angewandt: die Klassen *Link.java*, *Metric.java* und *Feed.java* stellen die grundlegende Struktur der Tabellen bereit, sowie eine Hilfsmethode, mit der Anfragen an die Datenbank abgesetzt werden können (siehe Listing 5.4 am Ende des Kapitels). Die eigentliche Funktionalität wird in den sogenannten „Agenten“ (*Feed-Agent.java*, *MetricAgent.java*, *LinkAgent.java*) definiert. Diese stellen die nötigen CRUD-Methoden zur Verfügung. Grund dieses Vorgehens ist die Kapselung von Model- und Controller-Logik. So soll in den Model-Klassen nur so viel Funktionalität wie nötig implementiert werden, alle für den Ablauf der Applikation kritischen Programmteile sollten ausgelagert werden. Die Agenten fungieren damit als Schnittstelle zwischen Applikation und Datenbank.

Wegen dieses Designs greift der Controller auf Lese- und Schreibmethoden grundsätzlich in der Form **RelationAgent.Methode()** zu, also etwa FeedAgent.createFeed(). So hat jede Relation ihren eigenen Agenten, der die für den Controller nötigen Operationen kapselt. Beinahe allen Methoden wird dabei stets - neben anderen Parametern - der Primärschlüssel der Tabelle übergeben. Im Falle der Metrik- und Feed-Relation ist dies ein eindeutiger Name, den der Anwender beim Speichern übergibt. Auf die Link-Relation wird über die Start- und Endknoten zugegriffen, deshalb genügt hier eine numerische ID als Primärschlüssel.

Die Applikation nutzt drei Relationen zur persistenten Speicherung, LINK, METRIC und FEED. Deren Struktur wird im folgenden erläutert.

FEED	
♦name	varchar(255)
•url	text
◦user	varchar(255)
◦pass	varchar(255)
◦x_path	varchar(255)
◦cached_value	int(11)
•last_refresh_at	datetime

Abbildung 5.2: Struktur der Tabelle FEED

METRIC	
♦name	varchar(255)
•unit	varchar(255)
•default_value	int(11)

Abbildung 5.3: Struktur der Tabelle METRIC

5.4.1 FEED-Relation

In dieser Klasse (Abbildung 5.2) sind alle Daten enthalten, die benötigt werden, um Feeds für die Applikation nutzbar zu machen. So werden neben einem eindeutigen Namen die URL des Feed-Dokuments und der zugehörige XPath-Ausdruck gespeichert, weiterhin ein *cached_ value*, der den Wert des letzten Abrufs enthält und ein Zeitstempel, der den letzten Zugriff auf den Feed angibt. Nutzername und Passwort werden im Klartext gespeichert, da eine sichere Implementierung nicht maßgeblich für den Ablauf war, zudem dürften die wenigsten Fälle eine Authentifikation erfordern.

5.4.2 METRIC-Relation

Diese Klasse speichert jede Metrik mit einem eindeutigen Namen, einer optionalen Einheit und einem Standardwert (siehe Abbildung 5.3). Die Einheit gibt dabei an, wie die numerischen Werte (*default_ value* bzw. Wert aus Feed) zu interpretieren sind, also etwa die Einheit „$“ für den aktuellen Kurs des Dollar. Zu beachten ist, dass der Einheit keinerlei Bedeutung im weiteren Ablauf zugemessen wird, sie dient lediglich als Anhaltspunkt für den Nutzer. Der Standardwert kommt für alle Netzkanten zu tragen, denen die entsprechende Metrik (identifiziert über den „name“) zugewiesen wurde; er wird überschrieben, falls einer Kante zusätzlich zur Metrik auch ein Feed zugewiesen wurde, nämlich mit dem Wert, den die Auswertung des XML-Dokuments ergibt.

5.4.3 LINK-Relation

Diese Klasse speichert Verbindungen zwischen zwei Knoten (Abbildung 5.4). Zu beachten ist dabei, dass niemals die gesamte Netztopologie in der Datenbank gespeichert wird, son-

[12] `http://www.avaje.org/ebean/introduction.html`

LINK	
♦id	bigint(20)
•source_interface	varchar(15)
•source_ip	varchar(15)
•target_interface	varchar(15)
•target_ip	varchar(15)
•subnet	varchar(15)
•metric_name	varchar(255)
•feed_name	varchar(255)

Abbildung 5.4: Struktur der Tabelle LINK

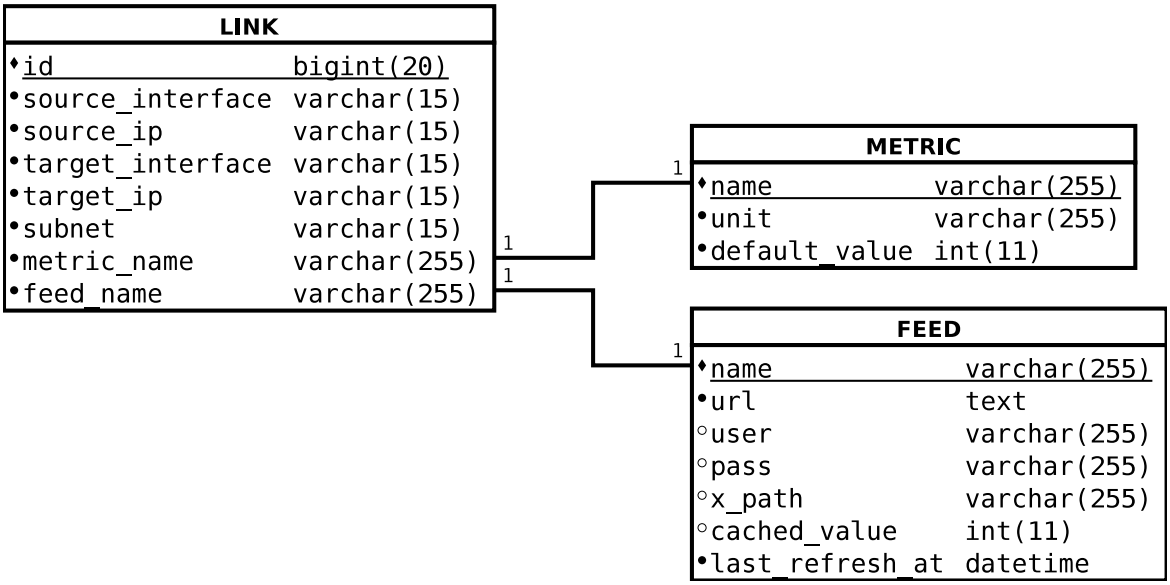


Abbildung 5.5: Beziehung zwischen Relationen

dern lediglich diejenigen Verbindungen, die vom Nutzer konfiguriert werden. Die Topologie selbst liegt im Hauptspeicher vor. Somit wird beim initialen Einlesen des Netzes noch nichts gespeichert; legt der Nutzer jedoch für eine oder mehrere Kanten einen Feed fest, wird diese Beziehung in der Datenbank gespeichert (auf die Beziehung der Relationen wird später noch ausführlicher eingegangen). Hintergund dieses Vorgehens ist zum einen die Ineffizienz eines vollständigen Speicherns der Topologie (Dynamik des Netzes, Serialisierung des Graphenobjekts), zum anderen das Bedürfnis des Nutzers, auf bereits angelegte Konfigurationen zurückzugreifen. Die Attribute *metric_name* und *feed_name* stellen dabei eine Fremdschlüsselbeziehung zu den jeweiligen Relationen her und referenzieren deren eindeutige „name“-Attribute.

Die Beziehung der einzelnen Relationen zueinander zeigt Abbildung 5.5. Einem Link ist stets eine und nur eine Metrik zugeordnet. Dieser Zuordnung können mehrere Feeds zuge-

wiesen werden. Folgendes Beispiel: Zwischen A und B existiert eine Kante. Dieser Kante ist die Metrik „sunshine hours“ zugewiesen. Ein Feed, der die Anzahl Sonnenstunden zu einem bestimmten Ort liefert, existiert. Dieser kann nun der bestehenden Beziehung zwischen Link und Metric zugewisen werden.

5.5 Feeds – Implementierung

Externe Datenquellen werden über Feeds erschlossen. Als Feed soll ein online verfügbares Dokument verstanden werden, das in einem maschinenlesbaren Format vorliegt und beliebige Daten bereit hält. Im Rahmen dieses Projekts wurde das Datenaustauschformat auf XML festgelegt, was eine Spezifizierung der gewünschten Werte über einen XPath-Ausdruck ermöglicht. Folgendes Beispiel soll dies verdeutlichen: bei Listing 5.2 handelt es sich um ein Dokument, das Wetterdaten zu europäischen Städten bereithält. Den XPath, der etwa die Sonnenstunden für Genf angibt, zeigt Listing 5.3.

```
<xml version="1.0" encoding="utf-8">
  <city name="Berlin">
    <rainfall in="ml per sqm">10</rainfall>
    <temperature in="°C">15</temperature>
    <sunshineHours in="hours">13</sunshineHours>
  </city>
  <city name="Geneva">
    <rainfall in="ml per sqm">0</rainfall>
    <temperature in="°C">25</temperature>
    <sunshineHours in="hours">14</sunshineHours>
  </city>
</xml>
```

Listing 5.2: Beispiel-Feed „sunshine hours“

```
/city[name="Geneva"]/sunshineHours/text()
```

Listing 5.3: Beispiel XPath-Ausdruck

Zur Anbindung externer Daten-Feeds werden also folgende Informationen vom Nutzer benötigt:

- URL des XML-Dokuments
- XPath-Ausdruck
- eindeutiger Name des Feeds in der Datenbank

Der Nutzer hat über die grafische Oberfläche die Möglichkeit, die URL sowie einen gültigen XPath-Ausdruck zu dem gewünschten Feed zu spezifizieren. Dieses Vorgehen mag fehleranfällig erscheinen im Hinblick auf komplexe XML-Dokumente, die einen ähnlich komplexen XPath bedingen, ist jedoch deutlich schlanker zu implementieren als eine „Point-and-Click“-Lösung.

Der Feed-Agent lädt beim Aufruf der Speichern-Routine zuerst die URL, unter der der XML-Feed verfügbar ist. Anschließend wird der XPath darauf angewendet, was schließlich in einem einzigen ganzzahlig positiven Wert resultiert. Dieser wird im Feld **cached_value** abgelegt, zusammen mit den vom Nutzer übermittelten Daten.

Danach hat der Controller jederzeit die Möglichkeit, den Wert, den ein Feed bereithält, aus der Datenbank abzufragen. Dies wird etwa beim Zuweisen eines Feeds zu einer Kante genutzt, wo der Wert die Kosten der Verbindung angibt.

Zum Datenaustausch würden sich neben XML auch JSON oder YAML eignen, wegen der weiten Verbreitung und der hervorragenden Unterstützung durch Java (SAX, `http://www.saxproject.org/`) wurde jedoch eine Beispielimplementierung auf Basis von XML gewählt. Ist die Nutzung eines anderen Formats gewünscht, sollte in *FeedAgent.java* ein Parser dafür implementiert werden. Die bestehende Lösung kann dabei problemlos ersetzt werden, sofern der resultierende Wert, den der Parser zurückliefert, ganzzahlig positiv ist. Da zur Routenberechnung der Algorithmus von Dijkstra benutzt wird, würden negative Werte hier zu Problemen führen.

Wird eine Unterstützung mehrerer Datenaustauschformate angestrebt, müssen auch andere Komponenten entsprechend angepasst werden. Beispielsweise sollte die Web-Oberfläche die Auswahl des Formats unterstützen, damit der Controller den jeweiligen Parser aufrufen kann; ggf. ist auch eine Anpassung des Datenmodells zu überdenken.

Für den Fall, dass der gewünschte Feed nur nach Nutzerauthentifikation verfügbar ist, prüft die vorliegende Implementierung das Protokoll der Verbindung, die den Feed abruft, und übergibt im Falle von HTTPS Authentifizierungsdaten. Diese Herangehensweise wurde gewählt, da im Rahmen der Entwicklung Feeds des Anbieters cosm.net[13] abgerufen wurden, auf die nur via HTTPS + HTTP Basic Authentication zugegriffen werden kann. Nutzername und Passwort werden vom Nutzer über die GUI spezifiziert und persistent gespeichert. Eine Lösung, die die Unterstützung von Basic Auth auch über HTTP erlaubt, wäre wünschenswert, wurde jedoch nicht im Rahmen dieses Projekts geleistet.

Die Feeds werden von der Anwendung in regelmäßigen Abständen aktualisiert (siehe hierzu 3.3.1) und die Werte, die der Parser dabei ausgibt, werden den Netzkanten als Kosten zugewiesen. Das bedeutet, dass der Kostenwert einer Verbindung nicht statisch ist, sondern sich, je nach Dynamik des Datenfeeds, ändert.

```
1  @Entity
2  @Table(name = "METRIC")
3  public class Metric extends Model {
4      @Id
5      @Column(unique = true, nullable = false)
6      private String name;
7      private String unit;
8      @Constraints.Required
9      private int defaultValue;
10
11     /*
12      * Getters for private fields
13      */
14     public final String getName() { return name; }
15     public final String getUnit() { return unit; }
16     public final int getDefaultValue() { return defaultValue; }
```

[13] mittlerweile in `https://xively.com/` aufgegangen

```
17
18      /*
19       * Setters for private fields
20       */
21      public final void setName(final String newName) { this.name =
        newName; }
22      public final void setUnit(final String newUnit) { this.unit =
        newUnit; }
23      public final void setDefaultValue(final int newDefaultValue) { this.
        defaultValue = newDefaultValue;  }
24
25      public Metric(final String theName, final String theUnit,
26              final int theDefaultValue) {
27          this.setName(theName);
28          this.setUnit(theUnit);
29          this.setDefaultValue(theDefaultValue);
30      }
31
32      /*
33       * Helper-method that assists in querying the DBMS
34       */
35      private static Finder<String, Metric> find = new Finder<String,
        Metric>(
36              String.class, Metric.class);
37
38      public static Finder<String, Metric> getFind() {
39          return find;
40      }
41  }
```

Listing 5.4: Modellierungsbeispiel *Metric.java*

6 AP4: Web-basierte Nutzerschnittstelle (von Miriam Popescu)

Die GUI vereint alle Backend-Informationen über das Netz und verarbeitet sie für den User. Außerdem stellt sie Eingabemöglichkeiten für userspezifisierte Metriken und Feeds zur Verfügung. Benötigte Daten werden direkt mit dem Arbeitspaket „Controller“ ausgetauscht (siehe Abbildung 6.1), der für die Kommunikation einen eigenen Service implementiert hat.
Die Anforderungen waren einfache Bedienung und klares Design. Die Benutzeroberfläche sollte alle Funktionen zur Verfügung stellen, die der User zur Verwaltung seines Netzes benötigt. Der Zustand des Netzes sollte auf Knopfdruck einsehbar sein und die Routen sollten nach nutzerdefinierten Kriterien angelegt werden. Daraus ergeben sich folgende Teilgebiete, auf die ich näher eingehen werde.

6.1 Teilgebiete

Design Anforderungen
Framework Play!
Datenaustausch Kommunikation über den *Communicator*
Darstellung Google Maps
Erweiterungsmöglichkeiten Mögliche Zusätze für die Applikation

Abbildung 6.1: Arbeitspaket GUI

6.2 Design

Es war wichtig, eine platformunabhängige und mobil einsetzbare Anwendung zu designen. Die Entscheidung fiel deswegen auf einer HTML-5 konformen Webapplikation. Dank des benutzten Twitter Bootstraps haben Eingabefelder und Buttons abgerundete Kanten, was für einen etwas „weicheren“ Look sorgt. Die Weboberfläche besteht aus einer Menüleiste und der Topologiedarstellung. Beim Aufruf der Index-Seite erscheint außerdem eine kurze Programmbeschreibung, um auch Neulingen einen leichten Einstieg zu gewähren.
Der User kann mit einem Klick auf „Get Topology“ sein Netz scannen. Wenn dieser Vorgang abgeschlossen ist, wird die Seite neu geladen, auf der Karte erscheinen alle Knoten und Links. Bei Klick auf eins der Knoten erscheint innerhalb der Karte ein Fenster mit Knotenspezifischen Infos wie Name und IP-Adresse. Über den Menüpunkt „Traceroute“ besteht die Möglichkeit, ein Traceroute zwischen zwei Punkten auszuführen. Möchte der Anwender nun eine neue Metrik oder Feed anlegen, muss er die dazugehörigen Formulare unter „New “ ausfüllen und sie einem Link zuweisen.

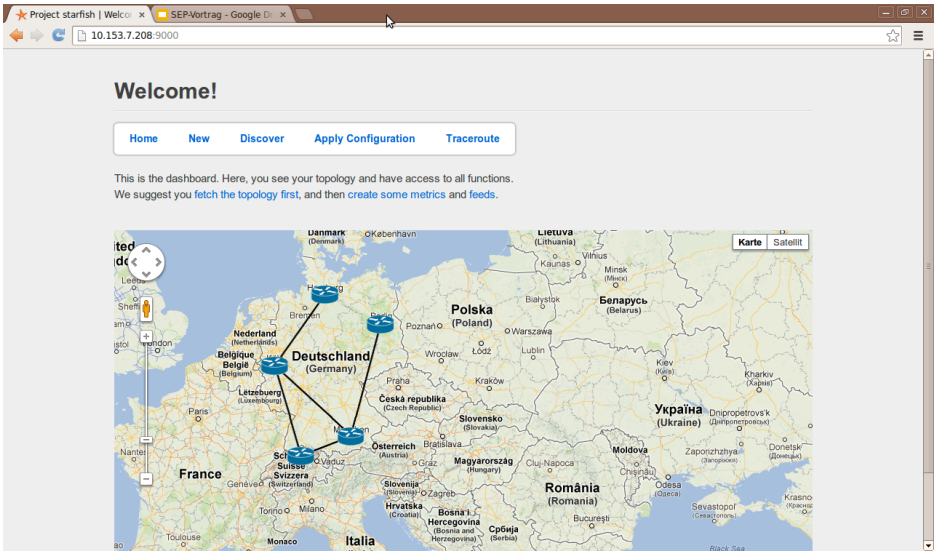


Abbildung 6.2: Benutzeroberfläche

6.3 Framework

Wie bereits erwähnt, haben wir für die Kommunikation zwischen Front- und Backend das play!-Framework (Version 2.0.1) benutzt[14]. Ein Vorteil war die schnelle und flexible Entwicklung, weil play! erst zur Laufzeit die Dateien generiert. Außerdem haben wir dadurch automatisch das MVC-Pattern ins Projekt integriert, das uns bei der Trennung der Aufgabenbereiche sehr geholfen hat.
Der Model-Teil ist in Zusammenarbeit mit Werner entstanden. Die benötigten Java-Vorgaben befinden sich im *app* Teil des Programms. Hier haben wir beschrieben, wie die GUI mit dem Rest des Backends zusammenarbeiten soll. An dieser Stelle befinden sich die Modellierun-

[14] `http://www.playframework.com/documentation/2.0.1/Home`

gen für unsere Metriken, Links und Feeds. Im Controller ist festgelegt, wie das Programm auf HTTP-Anfragen reagieren soll. Es werden Modelle gelesen oder aktualisiert und HTTP Antworten zurückgeschickt. In der *routes* Datei geben wir alle möglichen URL Aufrufe des Programms vor. Diese greifen mittels */GET* und */SET* auf einzelne Java Klassen zurück. Wenn die Antwort der Java Klasse *return ok* lautet, wird eine entsprechende HTTP Seite gerendert.
Des weiteren haben wir einen View-Bereich. Hier werden unsere Seiten mittels Scala beschrieben. Je nachdem, wie der Aufruf im Controller definiert wird, werden verschiedene Webseiten Templates aufgerufen. Läuft beispielsweise das Ausfüllen eines Formulars schief, wird eine Fehlerseite ausgegeben statt der normalen Webseite.

6.4 Datenaustausch

Die GUI stellt den View-Teil des Modells dar. Direkter Datenaustausch erfolgt mit Werners *Communicator* Teil, der von Werner geschrieben wurde. Die *Communicator.java* Klasse sammelt Informationen aus dem Netz und verarbeitet diese zu einem JSON-String. Hier sieht man, wie die modellierung des JSON Strings erfolgt. So erhält die GUI beim Aufruf der Klassen Informationen, die in Javascript weiterverarbeitet werden können.

```
1 s = s + "\"" + count + "\": {";
2 s = s + "üniqueID\": \"" + a.getUniqueIdentifier() + "\"" + ",";
3 s = s + "\"latitudeA\": " + a.getLatitude() + ",";
4 s = s + "\"longitudeA\": " + a.getLongitude() + ",";
5 s = s + "\"latitudeB\": " + b.getLatitude() + ",";
6 s = s + "\"longitudeB\": " + b.getLongitude();
7 s = s + " }";
```

Listing 6.1: Erstellung eines JSON Strings

6.5 Darstellung

Die grafische Darstellung der Routen wird durch die Implementierung der Google Maps JavaScript API v3 bewerkstelligt. Der Controller übergibt beim Aufruf der Java-Klassen, mittels .get, einen JSON String, der von einer Javascript Funktion geparst wird. Ebenfalls durch Javascript werden dann an den entsprechenden Koordinaten Icons für die Knoten gezeichnet. Meldet das Backend eine aktive Verbindung zwischen zwei Knoten, werden diese mit einer schwarzen Linie verbunden.

```
1 function getData(action, callback) {
2     var markerArray = [],
3     markerData,
4     marker;
5
6     $.get(action, function (response) {
7         var data = $.parseJSON(response),
8         i;
9
```

```
10          for(i in data) {
11              markerData = data[i];
12              marker = new google.maps.Marker({
13                  position: new google.maps.LatLng(
14                  markerData.latitude,
15                  markerData.longitude),
16                  icon: router,
17                  map: map
18                  });
19              markerArray.push(marker);
20              }
21          callback(markerArray);
22          });
23      }
```

Listing 6.2: Zeichnen von Knoten

6.6 Erweiterungsmöglichkeiten

Im Bereich der Usability wäre ein direkteres Feedback seitens der Google Map wünschenswert. Ist ein User beispielsweise gerade dabei, Metriken oder Feeds zu verändern, könnten auf der Karte die Links und Knoten sofort angezeigt werden, die voraussichtlich von dieser Aktion betroffen sind. Um den Ladezustand der Applikation deutlich darzustellen, wäre außerdem ein Overlay mit der Nachricht „Arbeite... Bitte warten “, nützlich.

Die Implementierung einer Rechtevergabe für Benutzeraccounts würde die Sicherheit und Stabilität des Netzes steigern. Es sind Szenarien denkbar, wo ein normaler User lediglich die Applikation lediglich zum Betrachten des Netzzustandes benutzt. Er sollte in diesem Fall keinen Zugriff auf Netzverändernde Formulare (z.B. Metriken, Feeds) haben.

7 Zusammenfassung

Die Idee zu diesem Projekt entstand im Februar 2012 aus dem Wunsch, den vorgegebenen Pfad, den die reguläre Veranstaltung „Softwareentwicklungs-Praktikum / System-Praktikum“ im dritten Bachelor-Semester der Informatik-Studiengänge vorgibt, zu verlassen. Die Autoren wollten eine eigene Software abfassen und nicht wieder „das Rad neu erfinden“ wie es viele Jahrgänge vor ihnen mit der Programmierung eines „Die Siedler“-Klons getan haben.
Mehr als ein Jahr später, im Juli 2013, ist das Projekt zu seinem Abschluss gekommen. In den vielen Monaten, die seit der ersten Kontaktaufnahme mit den zuständigen Professoren der LMU vergangen sind, haben alle Beteiligten enorm viel gelernt, sei es über Planung und Entwicklung eines Software-Projekts, über Programmier-Praxis oder über Selbstorganisation und Kommunikation im Team. Unser Dank gilt allen Personen, die dieses Projekt möglich gemacht haben. Rückblickend hätte vor allem das Zeitmanagement optimiert werden müssen. Hier ist der regulären Veranstaltung klar der Vorzug zu geben (16 Wochen gegenüber 73 Wochen). Allerdings konnte in diesem Projekt der gesamte Lebenszyklus einer Software, von der initialen Ideenfindung bis zur abschließenden Präsentation und Veröffentlichung, durchlaufen werden - eine Erfahrung, die uns sehr bereichert hat.

Bis zuletzt wurde versucht, eine stabile und lauffähige Anwendung zu erstellen. Vor allem wegen vieler Design-Annahmen (ausschließliche Unterstützung von Cisco-Geräten, geringdynamisches Netz, freie und risikofreie Verfügbarkeit beliebiger Daten-Feeds etc.) ist jedoch nicht auszuschließen, dass es zu Ausfällen oder Inkompatibilitäten im tatsächlichen Einsatz kommen kann. In der genutzten Testumgebung arbeiteten alle Komponenten nach Plan, allerdings ist fraglich, inwiefern diese Umgebung repräsentativ ist. Das eigentliche Ziel - intuitive Konfiguration der Wegewahl in Rechnernetzen anhand externer Daten - wurde jedoch erreicht. Es konnten Konfigurationen angelegt und auf die Netz-Knoten verteilt werden, woraufhin gezeigt werden konnte, dass eine Beeinflussung der Wegewahl möglich und im Sinne des Nutzers erfolgte. Folgendes Szenario diente im Rahmen des Abschlussvortrags als Test[15].

7.1 Abschließender Test

Für die Strecke München-Köln zahlt ein Unternehmen mit Standorten in allen deutschen Großstädten pro übertragenem Gigabyte einen variablen Preis, der sich täglich ändert. Die Preise dieses Tarifs können online über einen Daten-Feed abgerufen werden. Für alle anderen Strecken im Netz des Unternehmens gelten feste Preise. Es wird stets nach dem günstigsten Preis geroutet.
Beispiel: Sind die Strecken München-Zürich plus Zürich-Köln günstiger als die Strecke München-Köln, dann greift die Route via Zürich.

[15] Auf eine leichte Realitätsferne sei an dieser Stelle hingewiesen; der Test sollte lediglich Problem und Lösung deutlich machen.

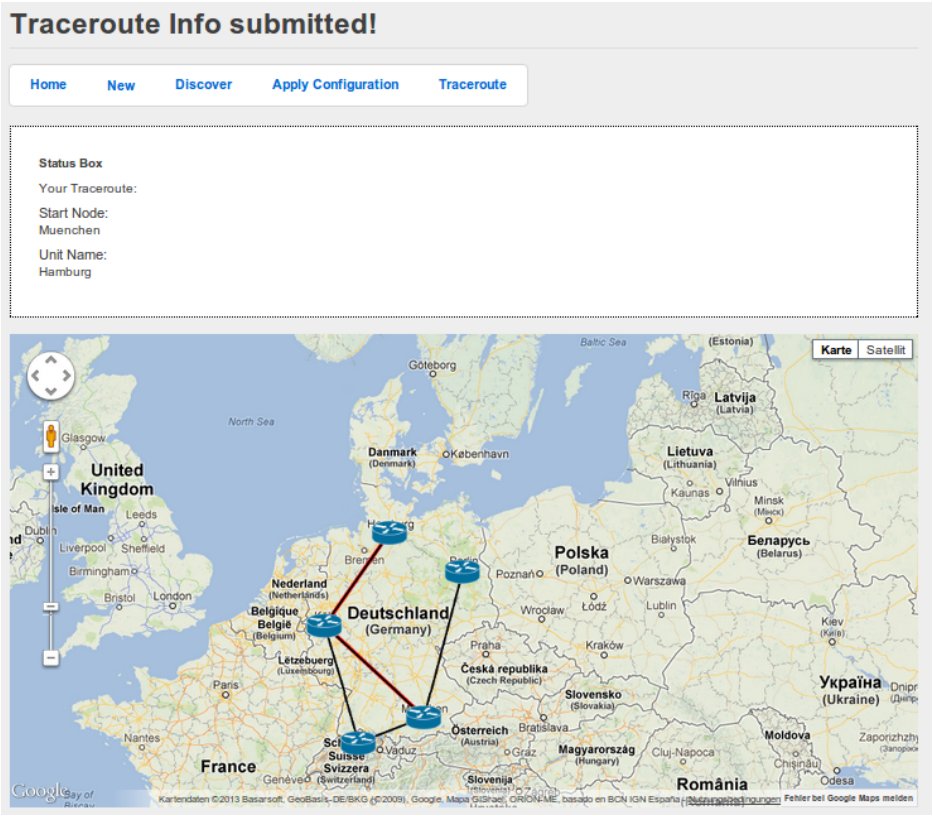


Abbildung 7.1: Initialer traceroute-Durchlauf

Ein initialer Durchlauf der traceroute-Funktion zeigt den Weg von München nach Hamburg über Köln (Abbildung 7.1).

Nach Anlegen einer Metrik „Preis“ (Einheit *€/GB*, Standard-Wert *10 €/GB*) und eines Feeds „MUC-COL“ (XPath */telekom/muc-col/price/text())* wurde für die Verbindung München-Köln die angelegte Metrik sowie der Feed konfiguriert (Abbildung 7.2).

Der erstellte Feed weist einen Kostenwert von 20 (€/GB) für die konfigurierte Verbindung aus. Somit ist der Weg von München nach Köln nun teurer geworden und der Weg über Zürich sollte bevorzugt werden. Ein weiterer Durchlauf von traceroute belegt dies (Abbildung 7.3).

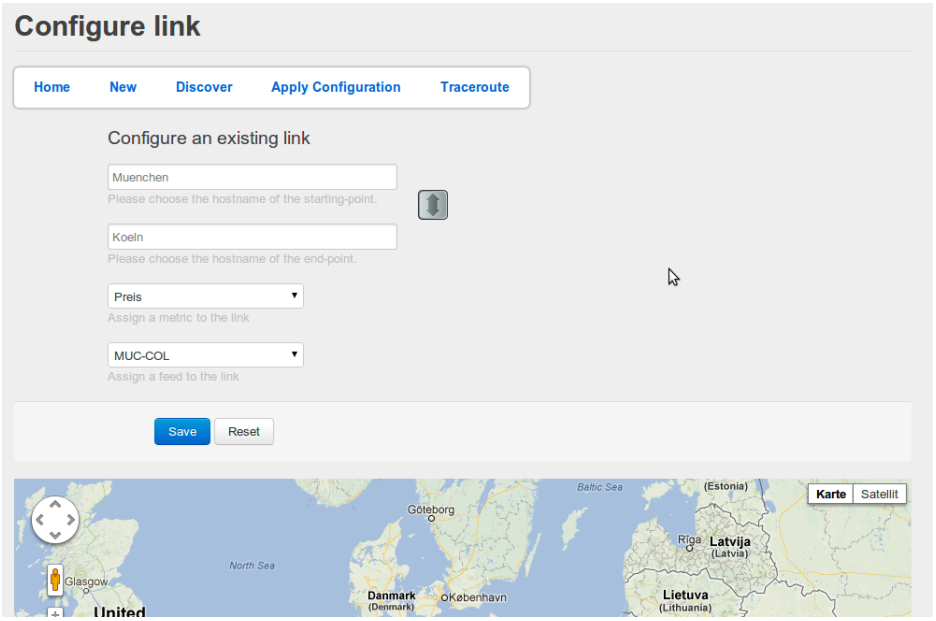


Abbildung 7.2: Konfiguration des Links

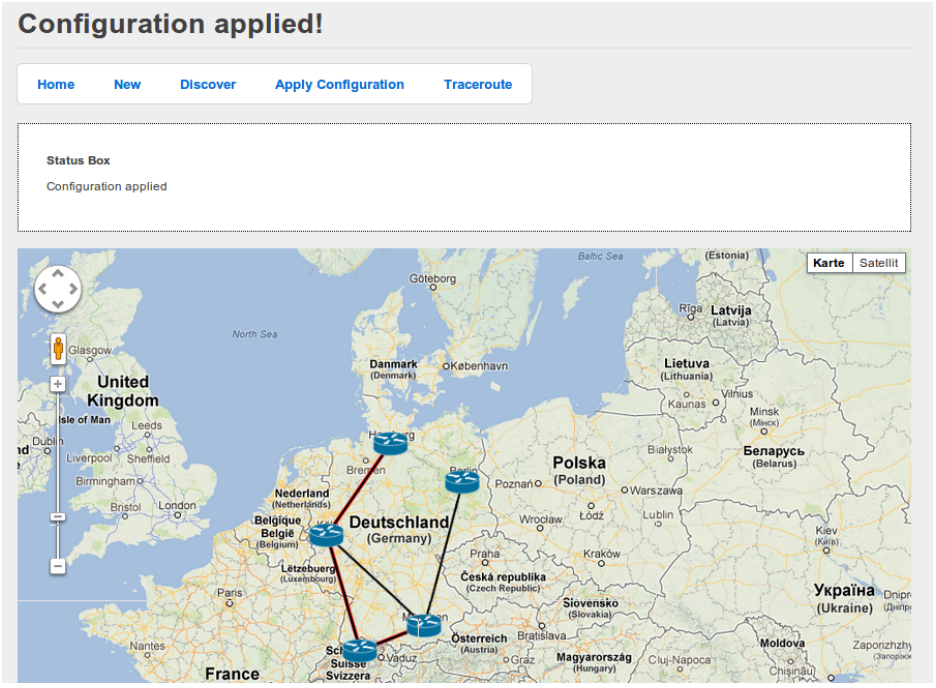


Abbildung 7.3: Abschließender Durchlauf von traceroute mit neuer Route

7.2 Schwächen und Verbesserungsmöglichkeiten

Trotz Erfüllung der gesteckten Ziele weist die Anwendung einige Schwächen auf. Etwa werden in der derzeitigen Version auf allen Netz-Knoten Routen gesetzt. Es wäre hier sinnvoller, die Routing-Tabellen auszuwerten und nur auf den relevanten Knoten statische Routen zu setzen. Außerdem benötigt die Arbeit der doWork-Methode (siehe Abschnitt 3.3.6) unverhältnismäßig viel Zeit. So dauerte der Ablauf im Test-Szenario (fünf Netzknoten) etwa 75 Sekunden. Hier sollte eine gründliche Performanz-Analyse erfolgen um diesen Ablauf auf ein Minimum zu reduzieren.

Im Netz-Modul wären verschlüsselte Verbindungen sehr zu begrüßen, da derzeit sämtliche Daten im Klartext übertragen werden. Hierzu zählt auch der Verbindungsaufbau zu den einzelnen Geräten, weshalb ein Angreifer ohne Mühe Nutzername und Passwort und damit Vollzugriff auf den Router beschaffen kann.

Das Feed-Modul würde enorm davon profitieren, wenn Änderungen von Daten nicht per Pull-Kommunikation regelmäßig abgefragt würden, sondern per Push direkt von der Quelle kämen. Ein Ansatz hierzu wäre eine Hardware-Demo mit einem Arduino oder RaspberryPi aufzusetzen, die dann in unmittelbarer Nähe eines Knotens aufgestellt wird. Sensoren oder Software würden Daten sammeln und öffentlich verfügbar machen; sobald es dann zu einer Änderung der Daten kommt könnte bspw. über eine dedizierte URL die Applikation informiert werden.
Außerdem wäre die Unterstützung verschiedener Datenaustauschformate denkbar (bislang nur XML).

Die web-basierte Nutzerschnittstelle sollte grafisch überarbeitet und mit einem individuellen Design versehen werden. Weiterhin lassen sich dank der Verwendung des Play-Frameworks hier Technologien wie LESS und CoffeeScript implementieren, da diese von Play automatisch in CSS bzw. JavaScript umgeschrieben werden. Schließlich sollte das Framework auf die aktuelle Version 2.1.1 aktualisiert werden.

Abbildungsverzeichnis