

Simon Tischler
Thomas Trügler

Anonymisierungsdienste Netzwerksimulator

BAKKALAUREATSARBEIT

zur Erlangung des akademischen Grades
Bachelor of Science

STUDIUM
Informatik

Alpen-Adria-Universität Klagenfurt
Fakultät für Technische Wissenschaften

BETREUER
Dr. Stefan Rass

Institut für Angewandte Informatik
Forschungsgruppe Systemsicherheit

Version 1.0

Klagenfurt, 12. August 2015

Bei Fragen, Problemen oder Anregungen kontaktieren Sie bitte die Forschungsgruppe Systemsicherheit (info@syssec.at) oder die Autoren (ttruegle@edu.aau.at, stischle@edu.uniklu.ac.at).

Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich, dass ich die vorliegende wissenschaftliche Arbeit selbstständig angefertigt und die mit ihr unmittelbar verbundenen Tätigkeiten selbst erbracht habe. Ich erkläre weiters, dass ich keine anderen als die angegebenen Hilfsmittel benutzt habe. Alle aus gedruckten, ungedruckten oder dem Internet im Wortlaut oder im wesentlichen Inhalt übernommenen Formulierungen und Konzepte sind gemäß den Regeln für wissenschaftliche Arbeiten zitiert und durch Fußnoten bzw. durch andere genaue Quellenangaben gekennzeichnet.

Die während des Arbeitsvorganges gewährte Unterstützung einschließlich signifikanter Betreuungshinweise ist vollständig angegeben.

Die wissenschaftliche Arbeit ist noch keiner anderen Prüfungsbehörde vorgelegt worden. Diese Arbeit wurde in gedruckter und elektronischer Form abgegeben. Ich bestätige, dass der Inhalt der digitalen Version vollständig mit dem der gedruckten Version übereinstimmt.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Simon Tischler
Thomas Trügler

Klagenfurt, 12. August 2015

Danksagung

Wir bedanken uns bei Assoc. Prof. DDipl.Ing.Dr. Stefan Rass und dem Team des Instituts für Systemsicherheit der Universität Klagenfurt für die Vergabe und Betreuung des Softwarepraktikums. Vor allem stand Dr. Rass uns immer wieder mit hilfreichen Anregungen zur Seite und lieferte in zahlreichen Gesprächen wichtige Denkanstöße.

Wir möchten uns auch bei Assoc. Prof. PD Dipl.-Ing. Dr. Peter Schartner für das Feedback innerhalb der Lehrveranstaltung und für zusätzliche Hilfestellungen bedanken.

Weiters gilt unser Dank Herrn Dr. Raphael Wigoutschnigg, da diese Vorlage für wissenschaftliche Arbeiten der Gruppe Systemsicherheit (syssec) auf den LaTeX-Files seiner Diplomarbeit aufbaut. Zudem stand er uns bei Beginn des Softwarepraktikums mit einer Einführung zum Thema “Simulation von Anonymisierungsprotokollen” mit Rat und Tat zur Seite.

Zusammenfassung

Jeder Mensch sollte auch im Internet einen gewissen Grad von Privatsphäre haben. Durch veröffentlichte Dokumente des Whistleblowers Edward Snowden wurde bekannt, dass die Öffentlichkeit von diversen Geheimdiensten abgehört werden kann. Doch es gibt Protokolle und Standards, die es Personen ermöglichen, ihre Korrespondenz zu verschlüsseln um das Lesen für Unbefugte unmöglich zu machen. Auch die Verschleierung der eigenen Identität ist mit Protokollen ebenso möglich.

Ziel des Softwarepraktikums, welches am Institut für Systemsicherheit durchgeführt und von Dr. Stefan Rass betreut wurde, war es einen Simulator zu schaffen, mit dem diverse Sicherheitsprotokolle einfach umgesetzt und ausgeführt werden können. So sollte eine empirische Untersuchung von Anonymisierungsprotokollen ermöglicht werden. Zudem war Teil der Anforderungen an den Simulator anderen Personen die Erweiterung um neue Protokolle zu ermöglichen. Die Sicherheit sollte experimentell überprüfbar sein und die Möglichkeit bestehen verschiedenste Angriffe gleichzeitig durchzuführen und zu untersuchen. Durch die Repräsentation der Nutzer eines Netzwerks mittels Knotenpunkte im Simulator werden die Angriffe simuliert.

So ist es möglich vielseitigste Angriffe zu erstellen und durchzuführen und so als Beispiel zu beweisen oder zu widerlegen, dass ein Protokoll Anonymität bietet. Die Entwicklung des Simulators orientierte sich hierbei am Protokoll "Crowds", welches auch als Demonstrationsbeispiel dient.

Inhaltsverzeichnis

1	Einleitung	1
2	Grundlagen	3
3	Algorithmen und Datenstrukturen	17
4	Implementierung	23
5	Beispielimplementierung	49
6	Simulationsablauf	55
7	Erfahrung	59
	Literaturverzeichnis	63

1 Einleitung

SIMON TISCHLER

Ziel des Softwarepraktikums war es, einen Netzwerksimulator zu implementieren, mit dem Anonymisierungsprotokolle einfach durchführ- und empirisch analysierbar sind. Anforderung war es, verschiedene Anonymisierungsprotokolle zu simulieren und Angriffe darauf durchzuführen, um so mögliche unbekannte Sicherheitslücken zu finden.

Der Simulator

Die Festlegung der Anforderungen für den Simulator erfolgte nach Rücksprache mit unserem Betreuer am Projektanfang. Die wichtigsten Anforderungen an den Simulator finden sich in der nachfolgenden Auflistung:

- Erweiterbarkeit des Simulators auch für Personen die nicht an der Entwicklung beteiligt waren
- Die Verwendung einer gängigen objektorientierten Programmiersprache für
- Einbeziehung von 2 Ebenen des OSI-Schichtenmodells
- Implementierung eines Routing-Algorithmus für die Berechnung von kürzesten Wegen im Netzwerk
- Inputparser der dem Nutzer die textuelle Beschreibung des Netzwerkes und der Protokollparameter erlaubt (Siehe Abschnitt 4).
- Detaillierte Dokumentation um weitere Protokolle im Simulator verwenden zu können. Siehe hierzu Kapitel 5 in dem Onionrouting als Beispielimplementierung verwendet wird.

Relevante Entscheidungen für die Implementierung, welche im Rahmen des Projektes getroffen wurden, waren:

- Der Entwurf von geeigneten Datenstrukturen zur Speicherung der Daten im Netzwerk
- Die Verwendung von Reflections, um den Typ von Objekten erst zur Laufzeit bestimmen zu können

Auf die Implementierung des Simulators und den oben genannten Details wird in Kapitel 4 näher eingegangen.

Motivation

Im Rahmen dieses Softwarepraktikums sollte nun ein Simulator für die empirische Überprüfung von diversen Anonymisierungsprotokollen implementiert werden. Mit Hilfe dieses Simulators ist es möglich, verschiedene Angriffe auf Protokolle zu simulieren. Somit ist es möglich Schwachstellen im Entwurf solcher Protokolle aufzuspüren, die zur Enttarnung von Nutzern führen können. Da neue Protokolle leicht überprüfbar sein sollten, lag das Hauptaugenmerk auf der Erweiterbarkeit. Das heißt zum einpflegen eines neuen Protokolles in den Simulator sollten keine Änderungen am Simulator selbst notwendig sein. Somit kann die einwandfreie Funktion des Simulators garantiert werden. Im Simulator wurde im Rahmen des Projektes, das Protokoll *Crowds* implementiert. Dieses Protokoll wurde in der zweiten Hälfte der 1990er Jahre von *Michael K. Reiter* und *Aviel D. Rubin* in den AT&T Labs entwickelt [7].

Anforderungen

Wie bereits erwähnt war das Ziel, die Software möglichst leicht erweitern zu können. Dadurch sollte es ermöglicht werden, ein neues Protokoll zu implementieren ohne Änderungen am bestehenden Programmcode vornehmen zu müssen. Weiters sollte der Simulator intuitiv und leicht zu verwenden sein. Die genaue Umsetzung dieser Anforderung findet sich im Kapitel 4. Zudem sollte die korrekte Funktionsweise des Simulators anhand einer Beispielimplementierung empirisch belegt werden.

Tools

Hier wollen wir kurz auf die verwendete Programmiersprache, Entwicklungsumgebung und andere verwendete Software eingehen.

Java

Java ist eine objektorientierte Programmiersprache. Sie ist Plattformunabhängig, das heißt, Sie kann auf nahezu allen wichtigen Prozessorarchitekturen ausgeführt werden. Weltweit gehört Java zu den am weitesten verbreiteten Programmiersprachen.

Frameworks

Verwendet wurde JavaFX für die Erstellung von graphischen Benutzeroberflächen und das Java Universal Network and Graph Framework (JUNG).

Weitere Tools

Weiters wurde für die Implementierung die Entwicklungsumgebung Eclipse verwendet. Diese ist leicht erweiterbar und bot den Studenten alle wichtigen Funktionalitäten. Zudem wurde Git verwendet, eine Versionierungssoftware (Version Control System), dass das bearbeiten von Dateien für mehrere Personen ermöglicht. Weiters wurde mit \LaTeX ein Typsatz-Software verwendet mit der die vorliegende Arbeit verfasst wurde.

2 Grundlagen

SIMON TISCHLER

In diesem Kapitel stellen wir grundlegende Technologien und Konzepte vor, die für das Verständnis der weiteren Arbeit nötig sind. Wir starten diese Einführung mit dem Thema Graphen, da diese die Basis von Netzwerken und dem Internet bilden. Dann geben wir einen kurzen Einblick die Kommunikation in Netzwerken, sowie einen Teil des zugrundeliegenden OSI-Schichtenmodells. Abschließend wollen wir noch auf Anonymität und das von uns gewählte Protokoll Crowds sowie zwei relevante Protokolle eingehen.

Graphen

Graphen sind Strukturen aus Knoten und Kanten. Wobei die Knoten als Objekte angesehen werden können. Die Kanten abstrahieren die Verbindungen zwischen den Knoten. Ein anschauliches Beispiel wäre die Darstellung von verschiedenen Städten oder Orten als Knoten und den Wegen zwischen ihnen als Kanten. Dies wird in Abbildung 2.1 nochmals veranschaulicht.

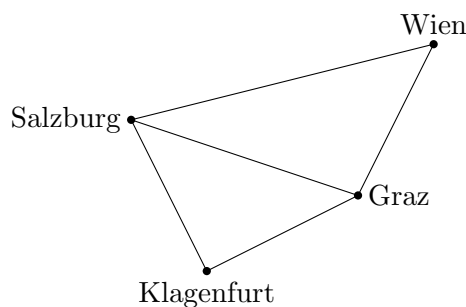


Abbildung 2.1: Beispielgraph mit Hauptstädten

Für die meisten Anwendungen von Graphen, vor allem im Bereich von Entfernungen ist es zudem wünschenswert, wenn die Entfernungen zwischen zwei Punkten, auf den ersten Blick erkennbar sind. Man spricht hier von *gewichteten* Kanten, es fallen also Kosten an, um von einem Knoten zu einem anderen Knoten zu kommen. Diese Kosten sind als Zahl auf der Kante eingetragen. Die Kante ist also gewichtet, dies ist in der Abbildung 2.2 ersichtlich.

Es werden zwei wichtige Arten von Graphen unterschieden, auf diese soll im Folgenden kurz eingegangen werden.

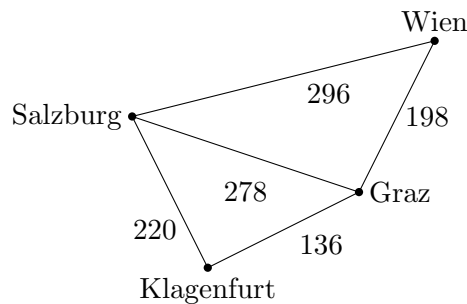


Abbildung 2.2: Österreichische Landeshauptstädte mit gewichteten Kanten, das Kantengewicht bezeichnet hierbei die Entfernung in Kilometern

Gerichtete Graphen

Hierbei sind die Kanten zwischen den Knoten gerichtet, dies wird durch einen Pfeil am Ende der Kante angegeben. Es wird also darauf hingewiesen, in welche Richtung die Kante passiert werden kann. Dies wird mit Abbildung 2.3 verdeutlicht. Hierbei sind die Wege nur in Pfeilrichtung passierbar:

- Klagenfurt nach Graz
- Klagenfurt nach Salzburg
- Salzburg nach Graz
- Salzburg nach Wien
- Graz nach Wien

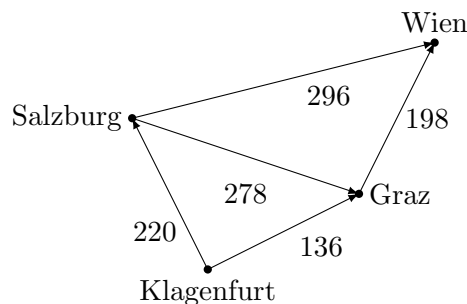


Abbildung 2.3: Hauptstädte mit gerichteten Kanten

Ungerichtete Graphen

In vielen Fällen, wie auch in Telekommunikationsnetzwerken ist eine Verbindung jedoch bidirektional, das heißt, eine Leitung kann in beide Richtungen genutzt werden. Im Falle des Internets ist dies leicht ersichtlich, da sowohl Daten aus dem Internet auf den Rechner geladen werden können, aber auch Daten vom Rechner ins Internet transportiert werden können. Deshalb sind diese Graphen für unsere Arbeit auch von größerer Relevanz und werden nun im Folgenden detaillierter erläutert.

Graph

Sei V eine endliche, nichtleere Menge von Knoten, sei E eine Teilmenge von

$$\{x, y\} \in V$$

die Kanten. So heißt $G = (V, E)$ ein *ungerichteter Graph*. [2]

Knotengrad

Sei $G = (V, E)$ ein Graph und $v \in V$ ein Knoten. Die Menge der Knoten, die durch eine Kante mit v verbunden sind, heißt die Nachbarschaft (neighbourhood) $U(v) := \{u \in V \mid \{u, v\} \in E\}$ von v . Der Grad (degree) von v bezeichnet die Größe der Nachbarschaft von v . Bezeichnung[2]:

$$g(v) := |U(v)|.$$

Darstellung von Graphen

Da in unserem Simulator ein Graph gespeichert und dargestellt werden soll, möchten wir hier einen kurzen Überblick über gängige Methoden zur Speicherung und Darstellung von Graphen geben. Es gibt prinzipiell zwei Arten, einen Graphen zu speichern. Entweder in Form einer Matrix, oder in Form einer Liste.

Matrix

Die Darstellung in Form einer Matrix kann auf zwei Arten erfolgen:

- Adjazenzmatrix: Hierbei handelt es sich um eine quadratische Matrix, bei der für jeden Knoten eine Zeile und eine Spalte existieren. Existiert nun eine Kante zwischen zwei Knoten, so wird in der Adjazenzmatrix für jeden Knoten in seiner Spalte eine 1 eingetragen, wenn eine Verbindung zum Knoten der entsprechenden Zeile besteht. Ansonsten wird eine 0 eingetragen. Bei dieser Art der Speicherung wird eine 0 für die Verbindung von einem Knoten zu sich selbst eingetragen. Wenn es sich um einen gewichteten Graphen handelt, so wird bei bestehenden Verbindungen anstelle einer 1 das tatsächliche Kantengewicht gespeichert.
- Inzidenzmatrix: In der Matrix wird für jede Kante eine Spalte und für jeden Knoten eine Zeile angelegt. Für eine gerichtete Verbindung von Knoten x zu Knoten y über die Kante e wird nun in der Spalte von e eine 1 in der Zeile für x eingetragen und eine -1 in der Zeile für y . Das Problem mit dieser Speicherungsart ist der Speicherverbrauch, da für jede Kante nur zwei Werte eingetragen werden und die restlichen Zeilen nur eine 0 speichern.

Liste

- Kantenorientierte Liste: In dieser Liste werden zunächst alle Knoten gespeichert und anschließend alle Kanten.
- Knotenorientierte Liste: In dieser Liste werden wiederum die Knoten zuerst gespeichert, wobei von einem Startknoten ausgehend sukzessive alle Nachbarknoten gespeichert werden. In dieser Liste werden wiederum zunächst alle Knoten gespeichert und anschließend sukzessive die direkten Nachfolger
- Adjazenliste: Bei dieser Form der Darstellung werden die Daten in einer Liste der Länge n gespeichert, wobei n die Anzahl der Knoten im Graph ist. Meist wird hierbei eine einfach

verkettete Liste verwendet, bei der jedes Listenelement wieder auf einen verketteten Speicher zeigt. Das heißt für jedes Listenelement L_0, L_1, \dots, L_{n-1} werden alle erreichbaren Knoten in einer weiteren Liste gespeichert.

Netzwerke & Internet

Die *Topologie* von Netzwerken, kann als Graph visualisiert werden. Die Kanten sind dann als Datenkabel oder auch kabellose Verbindungen wie Wireless LAN ausgeführt. Es wird angenommen das die Struktur des Internets dem Barabási-Albert Zufalls-Graphen-Modell entsprechend wächst (siehe Abschnitt 3). Weshalb dieses Modell für die automatische Generierung von Netzwerken in unserem Simulator verwendet wurde. k von Netzwerken. Das heißt, das Internet verbindet mehrere einzelne Subnetzwerke zu einem weltweiten Netzwerk. Für die Kommunikation in diesem Netzwerk werden verschiedenste Protokolle auf unterschiedlichen Abstrahierungsebenen benötigt. Diese Ebenen werden durch das OSI 7-Schicht Modell beschrieben. Die Subnetzwerke lassen sich in verschiedene Arten unterteilen, wobei hier vor allem die Position im Netzwerk betrachtet wird. Das Internet ermöglicht die direkte Verbindung zwischen zwei Computern in diesem globalen Netzwerk um Daten oder andere Informationen austauschen zu können.

Protokolle

Wenn in dieser Arbeit von Protokollen die Rede ist, so sind immer Protokolle gemeint, die sich mit der Anonymisierung von Nutzern beschäftigen. Diese zielen darauf ab, nicht nur den Inhalt einer Nachricht zu verschlüsseln, sondern auch Sender und Empfänger zu verschleiern. Dies ist oftmals ebenso wichtig, da ein Angreifer nicht immer nur an der Nachricht selbst interessiert ist, sondern auch daran, wer die Nachricht gesendet hat, wohin sie gesendet wurde und wann diese Kommunikation stattgefunden hat. Ziel der untersuchten Protokolle ist es also, die Anonymität des Senders im Netzwerk zu gewährleisten.

Begriffe und Hardwarekomponenten

Wir möchten nun einige Begriffe definieren, die bei der Erklärung eines Netzwerks wichtig sind. Zudem werden wichtige Hardwarekomponenten eines Netzwerks beschrieben.

Proxy Server

Bei Proxyservern handelt es sich um Schnittstellen zur Kommunikation in einem Netzwerk. Der entstehende Datenverkehr wird hierbei über einen anderen Knoten im Netzwerk gelenkt. Wird der Proxy als Komponente in einem Netzwerk eingesetzt, so wird die Identität des Nutzers welcher Daten anfordert gegenüber dem Webserver verborgen. Der Proxyserver kann die Pakete auch ändern. Es können auch Daten zwischengespeichert werden. Proxy-Server werden unter anderem für das Mix Protokoll verwendet (siehe Abschnitt 2).

Latenz

Latenz in Netzwerken ist jene Zeit, die eine Datenmenge benötigt um von einem Knoten zu einem Anderen zu gelangen. Da bei anonymisierungsprotokollen mehrere Knoten verwendet werden um die Nachricht zu versenden, kann die Latenz zwischen Sender und Empfänger sehr groß werden.

Routing

Routing bezeichnet die Vermittlung von Paketen innerhalb eines Netzwerks. Für das Routing können verschiedene Algorithmen eingesetzt werden. Idealerweise werden hierbei jene Algorithmen verwendet, die ein Datenpaket über den kürzesten Pfad im Netzwerk schicken (jener Pfad mit der geringsten Latenz). Abhängig von der Topologie des zugrundeliegenden Netzwerks können zwei Arten von Routing-Algorithmen unterschieden werden:

- Globale Topologie: Jeder Router kennt die gesamte Netzwerktopologie inklusive aller Latenzen. Es werden die sogenannten Link-state Algorithmen eingesetzt.
- Lokale Topologie: Bei dieser Variante kennt jeder Router nur seine direkten Nachbarn und die Latenzen dorthin. Der Prozess der Routenberechnung ist iterativ und für die Berechnung der kürzesten Wege ist ein ständiger Informationsaustausch zwischen den einzelnen Knoten nötig. Es werden Distance-Vector Algorithmen eingesetzt [9].

Es kann auch das Änderungsverhalten der Routen unterschieden werden. Eine Route muss geändert werden, wenn sich die Topologie des zugrundeliegenden Netzwerks ändert. Eine solche Änderung kann entweder durch den Ausfall eines Knotens oder durch das Hinzukommen eines neuen Knotens ergeben. Wenn ein Knoten ausfällt, über den Datenpakete versendet wurden, so muss ein neuer kürzester Weg berechnet werden um wieder Pakete übertragen zu können. Wenn ein neuer Knoten hinzukommt, so ist es möglich, dass über diesen Knoten Nachrichten günstiger versendet werden können. Das Änderungsverhalten ist abhängig vom verwendeten Algorithmus.

- Statisch: Statische Routen werden nachdem sie festgelegt wurden nurmehr geändert, wenn es zwingend notwendig ist, wie bei einem Knotenausfall.
- Dynamisch: Die Routen ändern sich in diesem Fall durch periodische Aktualisierungen oder durch Änderungen der Latenzen. Somit sind die Routen auch bei geänderter Netzwerktopologie am aktuellen Stand.

Das Routing passiert auf der Vermittlungsschicht, auf der die Pakete zwischen einzelnen Routern übertragen werden.

Das OSI 7-Schicht Modell

Das OSI-Modell ist ein Modell von Netzwerkprotokollen die in sieben Schichten, den sogenannten *Layern* aufgebaut sind und der Übertragung von Datenpaketen dienen. Jede Schicht hat eigene Aufgaben und ist logisch von den anderen Schichten abgegrenzt. Ein Vorteil des OSI-Modells ist, dass Protokolle der einzelnen Schichten einfach ausgetauscht werden können. Ziel ist es, Kommunikation zwischen unterschiedlichen technischen Systemen zu gewährleisten. Die Ebenen des Protokolls umfassen alle Teile einer Kommunikation.

Die Verwaltung und Abwicklung des Datenverkehrs wird meist durch das Betriebssystem vorgenommen, wodurch auch die Programmierung von Anwendungen erleichtert wird. Dies ist auch insofern sinnvoll, da für Benutzer nichts über die darunterliegenden Schichten bekannt sein muss. Für die Implementierung unseres Simulators waren zwei Schichten aus dem Modell von besonderer Bedeutung. Für das Protokoll wurden die Vermittlungsschicht (siehe 2) und die Anwendungsschicht (siehe 2) implementiert, also die Anwendungsschicht und die Vermittlungsschicht, die der Übertragung von Paketen im Netzwerk dienen. Dies ist für den Simulator besonders

interessant, da Angriffe auf zwei Schichten einfach Möglich sind. Auf der Anwendungsschicht können böswillige Clients implentiert werden, die nicht dem vereinbarten Protokoll folgen. Die Vermittlungsschicht, welche die Pakete über den kürzesten Weg im Netzwerk transportiert ist in unserem Simulator ein zentraler Angriffspunkt. Es gibt eine vielzahl von Szenarien für Angriffe, wobei vor allem *Denial of Service* (DoS) Angriffe und die Manipulation von Paketen von uns umgesetzt wurden. Details zu unserer Implementierung und den Angriffsszenarien werden in weiterer Folge noch ausführlich behandelt.

Vermittlungsschicht

Die Vermittlungsschicht ist auf OSI-Layer 3 zu finden und dient der Vermittlung von Datenpaketen im Internet, wozu Routing-Algorithmen verwendet werden. Zu den wichtigsten Aufgaben auf dieser Ebene des Modells zählt das Bereitstellen von Adressen, die im gesamten Netzwerk Gültigkeit besitzen und das Routen der Pakete durch das Netzwerk. Auch die Verwaltung von Routingtabellen die das Routing im Netzwerk erleichtern ist eine Kernaufgabe. Zudem muss auf die fehlerfreie Übertragung der Pakete zwischen den einzelnen Knoten geachtet werden. Hardware die sich auf dieser Schicht findet umfasst Router und Layer-3 Switches.

Anwendungsschicht

Die letzte Schicht im ISO OSI Modell ist die Anwendungsschicht. Auf ihr werden Funktionen für die Anwendungen zur Verfügung gestellt. Es werden Verbindungen zu den unteren Schichten hergestellt und die Dateneingabe und Datenausgabe finden hier statt.

Verschlüsselung

Verschlüsselung ist ein Vorgang, bei dem ein sogenannter Klartext mit einem Verschlüsselungsverfahren unverständlich gemacht wird. Das Ziel ist es, einen Angreifer oder eine andere unberechtigte Person, die in den Besitz der Daten kommt, diese nicht lesen kann. Oder zumindest so viel Aufwand in das Aufdecken der geheimen Nachricht investieren muss, dass dieser Aufwand nicht lohnen ist, Ein berechtigter Empfänger kann die Daten aber leicht entschlüsseln und damit wieder lesbar machen. Es werden grob zwei Arten der Verschlüsselung unterschieden. Einerseits gibt es *symmetrische Verschlüsselung*, bei der zwei Kommunikationspartner jeweils den gleichen Schlüssel verwenden. Da hier pro Kommunikationspartner ein Schlüssel geheim vereinbart oder ausgetauscht werden muss, wird für den Schlüsselaustausch häufig *asymmetrische Verschlüsselung* verwendet. Bei dieser Art der Verschlüsselung besitzt jeder Kommunikationsteilnehmer einen *öffentlichen Schlüssel*, der von anderen Teilnehmern verwendet werden kann um eine Nachricht zu verschlüsseln. Die Nachricht wird dann vom Nutzer mit seinem *privaten Schlüssel*, den nur er kennt, entschlüsselt.

Anonymisierung

Anonymisierung zielt darauf ab, einem Angreifer keine Hinweise auf den Initiator und den Rezipienten einer Nachricht zu geben. Völlige Anonymität ist im Internet quasi nicht möglich. Es ist jedoch zumindest möglich die Rückverfolgung für einen potenziellen Angreifer so schwer wie möglich zu machen. Pfitzmann und Waidner [6] beschrieben schon 1986 ein Modell, mit drei Arten der Anonymität.

Empfängeranonymität

Der einfachste Weg um völlige Empfängeranonymität zu erreichen, ist ein sogenannter Broadcast im Netzwerk. Das heißt, jeder erreichbare Knoten im Netzwerk erhält die selbe Nachricht. Um dennoch den korrekten Empfänger erreichen zu können, muss die Nachricht ein Attribut enthalten, welches sie für den jeweiligen Empfänger erkennbar macht. In diesem Zusammenhang wird auch von *impliziter Adressierung* gesprochen. Im Gegensatz zur expliziten Adressierung wird nicht die Adresse selbst zur Übermittlung verwendet, sondern in der Nachricht eine Adressierung stattfindet.

Unverkettbarkeit von Sender und Empfänger

Von diesem Grad der Anonymität wird gesprochen, wenn ein Angreifer einige potentielle Sender und Empfänger im Netzwerk kennt, aber keinen Empfänger einem Sender zuordnen kann. Ein Ansatz hierfür ist das sammeln von mehreren Datenpaketen in einem zentralen Server. Wenn genügend Datenpakete gesammelt wurden, können diese in geänderter Reihenfolge ausgesendet werden. Somit ist es für einen Angreifer nicht möglich nachzuvollziehen, welcher Sender mit welchem Empfänger in Kontakt steht.

Senderanonymität

Bei diesem Grad der Anonymität ist es einem Angreifer und insbesondere dem Empfänger nicht möglich, den Absender einer Nachricht zu identifizieren. Es kann dem Angreifer jedoch möglich sein, den Empfänger der Nachricht zu ermitteln.

Crowds

Crowds wurde von uns als Beispielprotokoll für die Implementierung des Simulators gewählt. Das Protokoll wurde von Reiter und Rubin 1997 veröffentlicht [7]. Der Name lässt sich mit “Menschenmassen” übersetzen, was auch die Kernidee des Protokolls beschreibt. Ziel ist es, die einzelnen Nutzer in einer “Menschenmasse” verschwinden zu lassen. Crowds bedient sich dabei einer Liste aller Teilnehmer. Diese wird vom Protokoll verwaltet und jede Person die das Protokoll nutzt wird in dieser Liste gespeichert. Die gesamte Kommunikation findet über zufällige Pfade durch das Netzwerk statt. Dadurch wird dem Angreifer die Erkennung des Initiators der Kommunikation erschwert.

Verbindungsaufbau

Für die Nutzung des Protokolls muss eine Verbindung zum Crowds-Server hergestellt werden. Sobald eine Verbindung aufgebaut ist, findet die gesamte Kommunikation nurmehr mittels des Crowds Protokolls statt. Allerdings ist es auch Angreifern möglich am Crowds-Server zu registrieren und so aktiv (zufällig) an der Kommunikation teilzunehmen. Es ist nicht möglich potenzielle Angreifer bei der Registrierung auszufiltern. Um dieses Problem zu beheben werden viele (vertrauenswürdige) Nutzer in der “Crowd” benötigt.

Funktionsweise

Sender- und Empfängeranonymität wird beim Crowds-Protokoll sichergestellt, indem zufällig eine Verbindung über mehrere Kommunikationsteilnehmer generiert wird. Wenn sich also ein

Benutzer anmeldet, wird er in die Liste der Benutzer in der “Crowd” hinzugefügt. Ab diesem Zeitpunkt wird auch er für die Übertragungen verwendet.

Wenn ein Paket unter Zuhilfenahme des Crowds-Protokolls versendet wird, wird es zunächst an einen zufälligen Knoten im Netzwerk gesendet. Es ist auch möglich, dass der Absender die Nachricht zunächst noch einmal an sich selbst sendet oder der Empfänger das Paket erhält. Der Knoten, der das Paket nun erhalten hat, sendet es mit der Wahrscheinlichkeit p_f an einen weiteren zufälligen Knoten in der Crowd. Mit der Wahrscheinlichkeit $1 - p_f$ wird das Paket an den letztlichen Empfänger gesendet. Die Wahrscheinlichkeit p_f bezeichnet hierbei die *probability to forward*, also die Wahrscheinlichkeit des Weiterleitens innerhalb der “Crowd”. Dieser Parameter wird vom Crowds-Server für alle Teilnehmer der Kommunikation festgelegt. Die Wahl der Größe der *probability to forward* hat signifikanten Einfluss auf die Sicherheit und die Dauer der Übertragung. Ein zu niedriger Parameter bedeutet, dass der Kommunikationsweg zum Empfänger zu kurz wird. Somit ist es wahrscheinlicher für Angreifer, den *Initiator* der Kommunikation, also jener Knoten, der das Datenpaket erstmalig versendet hat, ausfindig zu machen. Ein zu großer Parameter verlängert den Kommunikationsweg und somit die Übertragungszeit. Wenn ein Knoten eine Nachricht empfängt, wird von ihm eine Zufallszahl z erzeugt. Diese wird mit p_f verglichen. Wenn nun gilt $z \leq p_f$, so wird das Datenpaket an einen anderen zufälligen Knoten im Netzwerk gesendet. Dies wird so lange wiederholt, bis der zweite Fall eintritt, nämlich $z > p_f$, in diesem Fall wird nun das Datenpaket an den angegebenen Empfänger gesendet und die Übertragung beendet. Anzumerken ist, dass die Kommunikation innerhalb der Crowd verschlüsselt übertragen wird, wobei bei der Implementierung aus Gründen der Performanz und Einfachheit auf die Verschlüsselung verzichtet wurde.

Grade der Anonymität

Die oben genannten Eigenschaften der Anonymität werden nun um die Grade der Anonymität ergänzt. Diese wurden für die Analyse des Crowds Netzwerks von Reiter und Rubin eingeführt [7].

Unverdächtig (Beyond suspicion)

Dieser Status ist erreicht, wenn ein Angreifer zwar weiß, dass eine Nachricht gesendet wurde, er den Absender nicht mit einer Wahrscheinlichkeit, die besser als beim Raten ist, bestimmen kann.

Beweisbare Unschuld (Probable Innocence)

Der Sender ist beweisbar unschuldig, wenn es aus Sicht des Angreifers nicht beweisbar ist, dass der Sender auch der Initiator der Nachricht ist.

Mögliche Unschuld (Possible innocence)

Dieser Status ist erreicht, wenn eine *nichttriviale Wahrscheinlichkeit* besteht, dass der Sender der Nachricht, nicht der Initiator der Kommunikation ist (Bei *trivialer Wahrscheinlichkeit* wäre der Sender mit einer Wahrscheinlichkeit von 1 nicht der Initiator der Kommunikation).

Unterteilung der Angreifer

Angreifer haben mehrere Möglichkeiten, Anonymisierungsprotokolle anzugreifen. Zwei verbreitete Angriffsszenarien, die allgemein gegen Anonymisierung eingesetzt werden, sind im Folgenden angeführt.

Lokales Abhören

Hierbei befindet sich der Angreifer im selben Netzwerk wie der Initiator. Dies wäre bei einem Heimnetzwerk oder dem Intranet einer Firma möglich. Der Angreifer hat also kein Wissen über Kommunikationspartner, außer ihrer Adresse. Er kann aber alle Pakete die empfangen und versendet werden ebenfalls öffnen. Da die Kommunikation in der Crowd idealerweise verschlüsselt ist, sind die Daten für ihn nutzlos. Ein Angreifer kann ermitteln, ob eine Nachricht aus dem Netzwerk versendet wurde, indem er überprüft ob vorher eine Nachricht in das Netzwerk eingegangen ist. Crowds bietet keinen Schutz gegen eine solche Art des Angriffs. Da aber die Daten nicht im Klartext versendet werden und die Verbindung immer über mehrere Mitglieder der Crowd abläuft, kann ein Angreifer nur Nachrichten bestimmen die direkt aus dem Netzwerk gesendet werden und den direkten Vorgänger und Nachfolger einer Kommunikation bestimmen. Da ein solcher Angreifer Zugriff auf das Netzwerk des Betroffenen haben muss, handelt es sich hierbei meist um eine Person die sich unberechtigt Zugriff zu einem privaten oder firmeninternen Netzwerk verschafft hat. Bei diesem Angriff spielt der Wert p_f keine Rolle, da der Angreifer jene Nachrichten ermittelt, die aus dem internen Netzwerk versendet werden.

Betrügerischer Angreifer

Hier ist nun der Empfänger des Datenpaketes der Angreifer auf das Protokoll. In diesem Szenario kann die Empfängeranonymität nicht mehr gewährleistet werden. Durch diesen Angriff wird aber die Senderanonymität nicht geschwächt, da die Kommunikation weiterhin über die Crowds passiert. Somit ist es für jeden Knoten gleich wahrscheinlich, dass er der Initiator der Kommunikation zum Empfänger ist. Wichtig ist hier jedoch, dass keine personenbezogenen Daten im Paket enthalten sind. Für diese Attacke hat auch der Wert von p_f keinen Einfluss auf die Anonymität.

Angriffsszenarien im Crowds-Protokoll

Die folgenden *aktiven* Angriffsszenarien wurden von uns im Simulator implementiert und werden nachfolgend erläutert.

Beenden der Übertragung

Ein möglicher aktiver Angriff besteht in dem absichtlichen Nicht-Weitersenden von Nachrichten. Solche Angreifer werden auch als *Denial of Service* bezeichnet. Crowds stellt keine Mechanismen gegen diese Art von Angriff bereit. Wenn auf eine Nachricht aber nach einer gewissen Zeitspanne noch keine Antwort eingetroffen ist, wird vom Protokoll die Nachricht über einen anderen Knoten erneut übermittelt.

Änderung des Pakets

Ein Angreifer kann ein erhaltenes Paket vor der weiteren Übertragung auch abändern. Wenn die Nachricht verschlüsselt wurde, kann die Nachricht verändert werden, der Inhalt der Nachricht wird nach dem Entschlüsseln dadurch mit hoher Wahrscheinlichkeit nicht mehr sinnvoll sein. Bei unverschlüsselter Übertragung kann ein Angreifer jedoch den Inhalt des Pakets verändern. Um die Authentizität und Integrität des Pakets zu gewährleisten müssen in diesem Fall andere Vorkehrungen getroffen werden.

Kollaborierende Mitglieder

Diese Art von Angriff ist das einzige Angriffsszenario, dass spezifisch für das Crowds-Protokoll ist. Hierbei werden mehrere Computer die sich in der Crowd befinden von einem Angreifer oder einer Gruppe von Angreifern kontrolliert. Die Angreifer können nun Informationen austauschen die der Enttarnung des Initiators dienen. Zudem ist es den Angreifern möglich, ein anderes Protokoll zu verwenden, welches bewusst vom Crowds Protokoll abweicht.

Bei diesem Szenario ist vor allem interessant ob es dem Angreifer möglich ist, den Initiator der Nachricht zu identifizieren. Das heißt, es sind jene Pfade von Interesse, welche nicht von kollaborierenden Mitgliedern gestartet wurden, bei denen aber ein solcher Angreifer am Pfad erreicht wird. Das Ziel der Angreifer ist es also, den Initiator einer Kommunikation zu bestimmen. Wenn das Datenpaket keine identifizierenden Informationen enthält, kann ein Angreifer nur davon ausgehen, dass sein Vorgänger die Übertragung gestartet hat. Alle anderen Knoten die kein Angreifer dieses Szenarios sind, können nun natürlich auch der Initiator sein, die Wahrscheinlichkeit hierfür ist jedoch deutlich geringer.

Wir möchten nun analysieren [7], wie wahrscheinlich es ist, dass der Vorgängerknoten der Initiator der Nachricht ist. Hierzu soll H_k mit $k \geq 1$ den ersten Angreifer beschreiben. Dieser befindet sich an der k -ten Position in der Übertragung, in welcher der Startknoten die Position $k = 0$ inne hat. Natürlich kann dieser auch andere Positionen im Weg bekleiden. Diese sind wie folgt definiert:

$$H_{k+} = H_k \vee H_{k+1} \vee H_{k+2} \vee \dots$$

Das heißt, dass der Angreifer H_{k+} in der Kette der Kommunikation nach dem ausgewählten Knoten k vorkommt.

Sei nun I das Ergebnis, dass der erste Angreifer die Position $k = 1$ inne hat, also der direkte Nachfolger des Initiators ist. Somit folgt aus $H_1 \Rightarrow I$. Der Umkehrschluss gilt nicht, da der Initiator mehrmals am Weg vorkommen kann. Ziel der Angreifer ist $P(I|H_{1+})$ zu bestimmen, also die Wahrscheinlichkeit, dass der erste Angreifer tatsächlich der direkte Nachfolger des Startknotens ist. Weiters gilt, dass der Startknoten glaubhafte Unschuld gegenüber den Angreifern hat, wenn die Wahrscheinlichkeit kleiner oder gleich $\frac{1}{2}$ ist. Daraus ergibt sich auch folgende Definition:

Glaubhafte Unschuld

Der Startknoten einer jeder Kommunikation in der Crowd hat glaubhafte Unschuld, genau dann, wenn:

$$P(I|H_{1+}) \leq \frac{1}{2}$$

Um beweisbare Unschuld zu erreichen, muss folgendes gelten:

$$p_f \geq \frac{1}{2}$$

Weiters bezeichnet n die Anzahl aller Teilnehmer in der Crowd und c die Anzahl der kollaborierenden Angreifer zu jenem Zeitpunkt an dem eine Übertragung durch die Crowd gestartet wird (der Weg wird zu Beginn festgelegt, daher sind andere Angreifer zu vernachlässigen). Hieraus ergibt sich das Kerntheorem der Arbeit zum Crowds-Protokoll [7], dass auch mittels unserem

Simulator verifiziert wurde. Der Beweis des Theorems 5.2 kann der Arbeit [7] entnommen werden.

Überprüfung der Anonymität

Wir möchten nun auf das Theorem 5.2 aus der Arbeit zu Crowds eingehen.

$$Falls : n \geq \frac{p_f}{p_f - \frac{1}{2}} (c + 1) \quad (2.1)$$

[7] so hat der Initiator der Nachricht beweisbare Unschuld gegenüber den kollaborierenden Angreifern, wobei n die Gesamtzahl aller Knoten und c die Anzahl der kollaborierenden Mitglieder ist.

Um die korrekte Funktionsweise des Crowds-Protokolls empirisch überprüfen zu können, haben wir kollaborierende Angreifer in unserem Simulator implementiert (siehe Kapitel 4).

Für die Überprüfung der Beweisbaren Unschuld unter der Bedingung 2.1 wurden zweimal 100 Simulationen durchgeführt. Zuerst wurde die Voraussetzung des Theorems eingehalten, und danach wurde das Theorem verletzt. Wenn das Theorem verletzt wurde sollte es den Angreifern möglich sein, den Absender der Nachricht zu ermitteln.

Zur Überprüfung wurden jeweils 100 Simulationen mit je 200 Knoten durchgeführt. Für die Übertragungen wurde die $p_f = 0.7$ gewählt. Daraus ergibt sich bei 200 Knoten, dass $(c + 1) < 57.14$ sein muss um die Voraussetzung der Bedingung 2.1 nicht zu verletzen. Die Anzahl an kollaborierenden Knoten wurde mit 50, 55 und 58 Knoten für die drei Simulationsdurchläufe gewählt. Mit 50 Knoten wurde eine sichere Variante gewählt. Bei dieser wurde kein einziges mal der korrekte Absender überführt. Bei 55 Knoten wurde ebenfalls kein einziger Knoten enttarnt. Deshalb wurden weitere Versuchsreihen mit 56 und 57 Knoten durchgeführt. Auch bei diesen Simulationen war es den Angreifern nicht möglich den Absender mit einer Wahrscheinlichkeit größer $\frac{1}{2}$ zu enttarnen.

Knoten	Angreifer	#Simulationen	#Verletzt
200	50	100	0
200	55	100	0
200	56	100	0
200	57	100	0
200	58	200	100

Tabelle 2.1: Tabelle mit Angriffen auf das Crowds-Protokoll in unserem Simulator. Für alle Angriffe war die p_f mit 0.7 festgelegt. #Verletzt bezeichnet die Anzahl an Simulationen bei denen die Voraussetzung für das Theorem verletzt wurde.

Bei 58 Knoten hingegen wurde bei der Hälfte der 200 Simulationen der Absender enttarnt. Somit wurde gezeigt, dass beim Verletzen der Voraussetzung der Bedingung 2.1 der Absender mit einer Wahrscheinlichkeit von $\frac{1}{2}$ enttarnt werden kann.

Weitere Protokolle

Zum Abschluss dieses Kapitels, soll auf zwei weitere Anonymisierungsprotokolle eingegangen werden. Zum einen den Vorläufer des in dieser Arbeit beschriebenen Crowds-Protokolls, das sogenannte MIX-Netzwerk [3] und zum anderen das bekannte Onion Routing [8]. Im Kapitel 4 werden auch die Details zur Implementierung von Onion-Routing präsentiert um exemplarisch das einpflegen eines neuen Protokolls in den Simulator zu beschreiben.

Mix Netzwerk

Die Ideen zum Mix Netzwerk wurden erstmals von David Chaum im Jahre 1981 beschrieben [3]. Bei diesem Protokoll soll die Anonymität durch die Verwendung von Proxy Servern gewährleistet werden. Hierzu befindet sich im Netzwerk zusätzlich zur normalen Infrastruktur noch ein weiterer Rechner, der Mix. Nun wird eine Nachricht von einem Sender mit dem öffentlichen Schlüssel des Empfängers verschlüsselt, anschließend wird die Adresse des Empfängers an das Ende der Nachricht angehängt und anschließend die Nachricht nochmals mit dem öffentlichen Schlüssel des Mix-Rechners verschlüsselt. Dieser erhält die Nachricht, entschlüsselt Sie und entfernt die Adressinformationen wieder von der Nachricht. Die Nachricht wird gespeichert und gewartet bis genügend weitere Nachrichten eingelangt sind. Nun werden die Nachrichten in einer zufälligen Reihenfolge an die Empfänger gesendet. Diese können nun die Nachricht entschlüsseln und weiter verarbeiten. Das System ist sehr robust und kann Anonymität garantieren. Es muss jedoch vorausgesetzt werden, dass Nachrichten die mehrfach empfangen werden nur einmal versendet werden können, ansonsten kann ein Angreifer Nachrichten zuordnen. Wenn beispielsweise eine Nachricht doppelt an den Mix gesendet wird und nun zwei mal die identische Nachricht an den Empfänger weitergeleitet wird, ist die Anonymität des Senders nichtmehr gegeben. Dies kann aber durch eine Funktion die redundante Kopien von Paketen vor dem Versenden entfernt leicht behoben werden. Zudem kann ein Empfänger vom Mix signierte Bestätigungen erhalten, die belegen, dass ein Paket beim Empfänger eingegangen ist. Somit kann ein Sender nachweisen, dass die Nachricht vom Mix nicht ordnungsgemäß übermittelt wurde. Mix Netzwerke können aber auch mehr als einen Mix enthalten und somit kann ein Pfad über mehrere solcher Server erstellt werden um die Sicherheit weiter zu erhöhen. Mix ermöglicht es dem Empfänger aber auch, dem Sender zu antworten und dabei weiterhin die Anonymität des Senders gegenüber dem Empfänger zu gewährleisten. Dies geschieht mittels einer Rücksenderadresse, die nicht rückverfolgt werden kann.

Onion Routing

Das wohl bekannteste Protokoll zur Anonymisierung in Netzwerken ist das sogenannte Onion Routing [8]. Bekanntheit erreichte es vor allem, da es bereits im Internet ausgerollt wurde und als Tor-Netzwerk bereits genutzt werden kann um sicher im Internet zu surfen. Die Nutzung des Netzwerks kann über einen vorkonfigurierten Browser oder über manuelle Konfiguration erfolgen. Die Grundlegende Funktionsweise von Onion Routing ist ähnlich zum Crowds-Protokoll, wobei jedoch nicht in jedem Knoten bestimmt wird, welcher Knoten als nächstes das Paket erhält. Beim Onion Routing wird zu Beginn einer Übertragung die gesamte Route vom Knoten bestimmt. Der Benutzer baut eine Verbindung zum Onion-Netzwerk auf und erhält eine Liste von verfügbaren Knoten im Netzwerk. Aus diesen wird der Pfad entlang dessen die Nachricht verschickt wird. Zudem enthält die Liste der verfügbaren Knoten auch die öffentlichen Schlüssel

der Knoten. Wenn der Pfad bestimmt wurde wird mit diesen in umgekehrter Reihenfolge des Pfades verschlüsselt. Begonnen wird die Verschlüsselung also mit dem letzten Knoten entlang des Pfades. Am Ende wird mit dem öffentlichen Schlüssel des Knotens der als erstes die Nachricht erhält verschlüsselt. Das Paket ist also Schichtweise aufgebaut und kann mit einer Zwiebel verglichen werden. Der erste Knoten in der Liste erhält dann die Nachricht und entschlüsselt Sie. Er entfernt also die äußerste Schicht der Verschlüsselung und schickt das Packet weiter. Dies ist wiederum vergleichbar mit dem entfernen einer Schicht der Schale eines Zwiebels. Daher leitet sich auch der Name dieses Protokolls ab. Mit dem Paket wird in jede Schicht auch der Nächste Knoten mitverschlüsselt. So ist auch der Nachfolgerknoten immer nur für den Knoten der die Nachricht erhält identifizierbar. Das bedeutet, dass jeder Knoten eine Schicht entschlüsselt, den nächsten Empfänger identifiziert und die Nachricht an diesen weiterleitet. Durch die mehrfache Ver- und Entschlüsselung mit dem eingebetteten Nachfolgerknoten, ist es den einzelnen Knoten auch nicht möglich, den Weg der Nachricht nachzuvollziehen. Zudem können die einzelnen Knoten auch nicht bestimmen wie lang der Pfad ist, entlang dessen das Paket versendet wird. Lediglich der Empfänger der Nachricht, kann seine Position entlang des Pfades bestimmen. Die Antwort wird vom Empfänger wiederum entlang des selben Pfades an den Sender zurückgeschickt, wobei das Paket nun natürlich in umgekehrter Reihenfolge ver- bzw. entschlüsselt wird. Das hierfür erforderliche Antwortpaket muss vom Sender der Nachricht vorab bereitgestellt (und mitgeschickt) werden.

Auch die Schwächen des Protokolls sind ähnlich zu Crowds. Onion Routing setzt keine Maßnahmen um zu verhindern, dass Nachrichten direkt im Netzwerk des Senders abgehört werden. Ein wesentlicher Unterschied ist, dass Onion-Routing Sender und Empfänger schützt Crowds jedoch nur Senderanonymität gewährleistet. Zudem hat das Protokoll zwar Maßnahmen gegen die Analyse des Webverkehrs, kann aber die Bestätigung des Webverkehrs nicht verhindern.

Wie bereits erwähnt, wurde das Protokoll nicht implementiert, im Kapitel 4 werden wir jedoch darauf eingehen wie dieses Protokoll in unserem Simulator implementierbar wäre.

3 Algorithmen und Datenstrukturen

SIMON TISCHLER

In diesem Kapitel soll auf die zugrundeliegenden Algorithmen und Datenstrukturen eingegangen werden. Diese bilden die Basis für die Implementierung unseres Simulators. Die verwendeten Datenstrukturen wurden von uns teilweise selbst konzipiert.

Algorithmen

Algorithmen sind eindeutig beschriebene Lösungsvorschriften für verschiedenste Problemstellungen. Im Simulator wurden zwei Algorithmen implementiert. Einerseits ein Algorithmus um kürzeste Wege in einem Graphen zu finden, andererseits ein Algorithmus um automatisch ein Zufallsnetzwerk zu generieren, dessen Topologie mit der des Internets “vergleichbar” ist.

Dijkstra-Algorithmus

Für den Simulator wurde von uns – nach Absprache mit unserem Betreuer – der Dijkstra-Algorithmus gewählt. Dieser war uns bereits aus dem Kurs “Algorithmen und Datenstrukturen” des Instituts für Systemsicherheit bekannt und ist leicht implementierbar. Zudem verfügt er über gute Laufzeiteigenschaften.

Funktionsweise

Der Dijkstra-Algorithmus gehört zur Familie der *Greedy*-Algorithmen.

Diese Algorithmen zeichnen sich dadurch aus, dass Sie in jedem Schritt immer den besten Wert auswählt. Er wählt also das lokale Optimum, und hofft, dass diese Auswahl zur besten globalen Lösung führt. [4]

Der Dijkstra-Algorithmus wurde nach seinem Erfinder dem niederländischen Mathematiker Edsger W. Dijkstra benannt.

Die Funktionsweise lässt sich nun wie folgt beschreiben: Der Algorithmus erhält einen Start- und einen Endknoten sowie das zugehörige Netzwerk. Nun wird für jeden Knoten ein Tripel gebildet. Diese hat die Form :

(Aktueller Knoten, Kosten bis zum Startknoten, direkter Vorgängerknoten)

Für alle Knoten im Netzwerk werden diese Tripel gebildet, wobei der Startknoten das Tripel (Startknoten, 0, Startknoten) hat, der Startknoten hat den kürzesten Weg mit Kosten 0 zu sich selbst und ist sein direkter Vorgänger. Im weiteren Verlauf des Algorithmus wird nun in jedem Schritt ein neuer Knoten zur Menge der gewählten Knoten hinzugenommen. Dieser muss

ein Randknoten sein, direkt mit einem der schon gewählten Knoten verbunden sein. Wenn am Ende alle Knoten im Netzwerk erreicht sind, lassen sich über die Vorgängerknoten - die in den Tripeln gespeichert sind - die kürzesten Wege im Graphen ablesen. Zudem lassen sich so auch die Gesamtkosten des kürzesten Weges ablesen.

Komplexität

An dieser Stelle möchten wir nun kurz auf die Komplexitätsklasse des Algorithmus eingehen.

Für die Bestimmung der Komplexität benötigen wir nun

E ... die Anzahl der Kanten im Graphen

und

V ... die Anzahl der Knoten im Graphen

Alle Knoten die noch nicht abgearbeitet worden sind, befinden sich nun in einer Warteschlange. Bei jedem Durchlauf der Schleife, wird ein Knoten entfernt. Wenn der Algorithmus aufgerufen wird, befinden sich in der Warteschlange V Knoten. Nun muss in jedem Durchlauf der entsprechende Knoten gefunden und entfernt werden. Wenn – wie zumeist üblich – die Warteschlange mit `PriorityQueues` implementiert wurde, wird der Knoten in $\mathcal{O}(\log V)$ entfernt. Somit benötigt das entnehmen aller Knoten aus der Schleife $\mathcal{O}(V \log V)$ Operationen.

Weiteres werden in einer Methode die die Randknotenmenge erweitert alle von den bereits gewählten Knoten aus erreichbaren Knoten betrachtet. Also alle Kanten die einen gewählten und einen nicht gewählten Knoten verbinden. Somit ergibt sich auch hier eine Komplexität von $\mathcal{O}(E \log V)$.

Daraus resultiert eine Gesamtkomplexität für den Algorithmus von:

$$\mathcal{O}(V \log V + E \log V).$$

Wahl des Algorithmus

Das Routing, welches einen zentralen Teil unseres Simulators bildet, findet auf der Netzwerkschicht statt. Diese ist eine der beiden erwähnten Ebenen die im Simulator implementiert wurden. Beim Routing wird ein Datenpaket, das von einem Knoten zu einem anderen gesendet wird über eine physische Verbindung im Netzwerk gesendet. Bevorzugt wird die günstigste Verbindung zwischen zwei Knoten gesucht, jene Verbindung mit der kürzesten Übertragungsdauer. Am Beispiel eines Graphen bedeutet dies, dass die Kosten zwischen den Knoten die benötigte Übertragungszeit darstellen. Wir haben uns im Simulator für den Dijkstra-Algorithmus entschieden. Dieser gehört zur Gruppe der *Link-State Algorithmen* [9]. Jeder Knoten der sich im Netzwerk befindet kennt die gesamte Topologie des Netzwerkes und kann so die kürzesten Wege zu allen anderen Knoten berechnen. Das heißt auch, dass der Dijkstra-Algorithmus viele Verbindungen berechnet die momentan nicht benötigt werden, diese werden gespeichert und können für spätere Übertragungen im Netzwerk genutzt werden. Zudem war die Verwendung des Dijkstra-Algorithmus leichter umsetzbar, da ohnehin schon die gesamte Netzwerktopologie gespeichert wird.

Der Algorithmus wurde auch in Lehrveranstaltungen behandelt und ist auch vergleichsweise einfach zu implementieren. Leider nahm die Implementierung trotzdem etwas mehr Zeit in Anspruch als Angenommen, was aber unseren Datenstrukturen geschuldet ist.

Alternativ zum Dijkstra-Algorithmus hätte auch der Algorithmus von Floyd-Warshall implementiert werden können. Dieser hat jedoch schlechtere Laufzeiteigenschaften, da für jedes Knotenpaar im Graphen der kürzeste Weg berechnet wird. Bei einer einzigen Kommunikation in einem größeren Netzwerk wären somit sehr viele Wege umsonst berechnet worden. Zudem hätte der Algorithmus schon beim Starten des Simulators verwendet werden müssen. Bei größeren Netzwerken wäre dies aber nicht sinnvoll gewesen, da so bis zu einigen Minuten an preprocessing nötig gewesen wären, bis die Simulation der Übertragungen begonnen hätte.

Die Verwendung des Dijkstra-Algorithmus findet bei uns auch wegen des unnötigen "preprocessing on demand" statt. Das heißt der Algorithmus wird aufgerufen, sobald eine Übertragung auf der ISO-OSI Schicht 7 benötigt wird. Nun wird die Schicht 3 des Netzwerkes simuliert und hier der kürzeste Weg im Netzwerk vom Startknoten zum Endknoten berechnet. Jedes mal wenn ein Endknoten erreicht wird, wird zufällig gewählt ob zum Endknoten oder zu einem Anderen Knoten gesendet werden soll. In der Praxis hat sich hier schon gezeigt, dass bei Netzwerkgrößen über 2000 Knoten bereits hier mit längeren Wartezeiten bei der Berechnung eines kürzesten Weges zu rechnen ist.

Barabási-Albert model

Der zweite Algorithmus der im Simulator Verwendung fand ist das sogenannte "Barabási-Albert Model". Es dient zur Generierung von Netzwerken und wird besonders wegen seiner realistischen Abbildung des Internets verwendet, da Knoten die mehrere Verbindungen haben, eher weitere erhält. Für jene Knoten mit weniger Verbindungen ist die Wahrscheinlichkeit geringer.

Funktionsweise

Der Algorithmus basiert im wesentlichen darauf, dass Knoten die schon viele Verbindungen haben, weitere dazu bekommen und Knoten mit wenigen Verbindungen nicht weiter verbunden werden. Für die Generierung von Netzwerken mit bis zu 100 Knoten wird im Simulator jedoch nicht der oben beschriebene Algorithmus verwendet. Hier erfolgt die Erzeugung des Netzwerkes zufällig für jedes Knotenpaar.

Der Algorithmus wird anhand der Formel:

$$p_i = \frac{k_i}{\sum_j k_j}$$

entwickelt. Wobei p_i die Wahrscheinlichkeit darstellt, dass ein neuer Knoten mit dem Knoten i verbunden wird. Weiters ist k_i ist der Grad vom Knoten i und die Summe wird über alle bereits existierenden Knoten j gebildet. Somit erhalten besser vernetzte Knoten eher neue Knoten hinzu [1].

Datenstrukturen

Wie bereits erwähnt wurde bei der Implementierung des Simulators auf den Einsatz von Kanten zur Modellierung des Graphen verzichtet. Stattdessen wird eine Adjazenzliste zur Speicherung der Datenstrukturen verwendet. Das heißt in jedem Knoten sind alle seine Nachfolgeknoten inklusive ihrer Latenz gespeichert. Mit zunehmender Anzahl an Verbindungen zwischen den Knoten, steigt zwar der Speicherverbrauch, dieser ist aber für unsere Anwendung vernachlässig-

bar. Zudem wurde aus Performanzgründen eine obere Schranke für die Anzahl an Verbindungen pro Knoten gesetzt, die auch den Speicherverbrauch gering halten.

ReachableNodes

Ein `ReachableNode`-Objekt besteht aus einem Knotenobjekt und einem Double-wert für die Latenz zwischen den Knoten. Für jeden Knoten wird für Verbindung ein Objekt erzeugt und gespeichert.

Wie in jedem Netzwerk gibt es auch in unserem Simulator unterschiedliche Latenzen auf den Leitungen. Somit speichert ein `ReachableNodes`-Objekt die Identifikationsnummer des Knotens und zudem die erwähnte Latenz. Die `ReachableNodes` werden in `ReachableLists` verwaltet, auf die wir in Abschnitt 3 näher eingehen.

Für den Simulator waren auch die Latenzen von großer Bedeutung, da der Netzwerksimulator so realistisch wie möglich sein sollte. Daher wird für jedes Listenelement nicht nur die bestehende Verbindung gespeichert, sondern auch die Latenz der bestehenden Verbindung.

ReachableList

Die `ReachableNodes`, die die Randknoten im Dijkstra-Algorithmus symbolisieren, werden in unserem Simulator nun in der `ReachableList` verwaltet. Hierfür wird für jeden Knoten eine `ReachableList` angelegt. In dieser Datenstruktur auf die leicht zugegriffen werden kann sind nun alle Verbindungen gespeichert und können abgerufen werden.

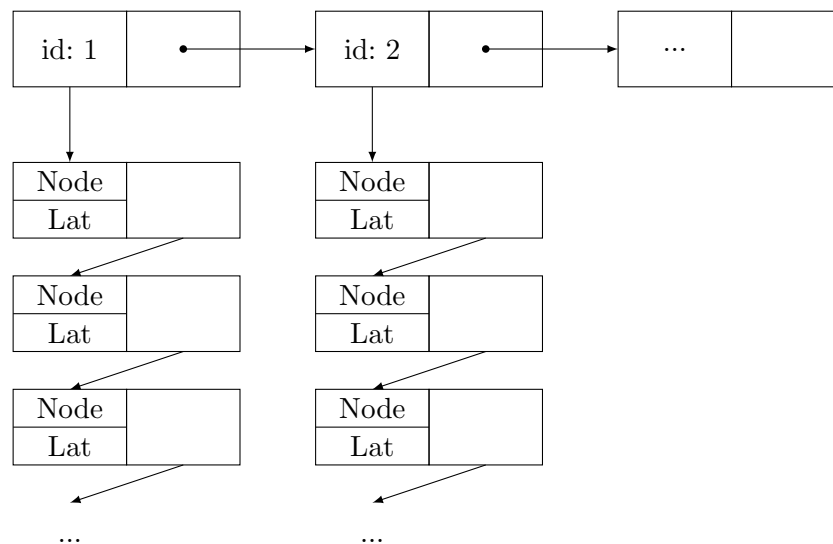


Abbildung 3.1: Veranschaulichung der `ReachableList` der die Datenstruktur einer Adjazenzliste zugrunde liegt

Ein `ReachableList` Objekt entspricht als Datenstruktur einer Adjazenzliste. Dies entspricht einer Verketteten Liste, bei der jedes Listenelement auf eine weitere verkettete Liste mit den erreichbaren Knoten zeigt. Die Datenstruktur wird zudem nochmals in der Abbildung 3.1 dargestellt. Die obere Reihe der Knoten die horizontal verbunden sind, stellt die Liste aller Knoten dar. Da es sich um eine verkettete Liste handelt zeigt jedes Objekt auf das Nachfolgeobjekt.

Zudem zeigt jedes Element auf eine weitere verkettete Liste. Diese symbolisiert die erreichbaren Knoten und besteht aus einem **ReachableNode**-Objekt, dass neben der eindeutigen Identifikationsnummer noch die Latenz der Verbindung speichert. Zudem wird in jedem Element noch ein Zeiger auf das nächste Listenelement verwaltet.

4 Implementierung

THOMAS TRÜGLER

In diesem Kapitel möchten wir näher auf die Implementierung des Netzwerksimulators eingehen. Diese werden wir durch Erläuterung der Projektstruktur und Organisation der Klassen und Logik näher bringen. Die Analyse der Funktionalitäten, die Umsetzung der Datenstrukturen und die Verwendung von Algorithmen erschließen weitere Teile des Simulators. Durch konkrete Codeausschnitte werden die Kernfunktionen beschrieben werden. Der Simulator wurde mit *Java* in Kombination mit den Frameworks *JavaFX* und *JUNG* entwickelt. Ein zentraler Aspekt des Netzwerksimulator ist seine Erweiterbarkeit (siehe Abschnitt 4). Diese wird auch durch die bewusste Definition und Verwendung von Interfaces und allgemeinen Methoden ermöglicht. Für die Umsetzung der Erweiterbarkeit und um die Möglichkeiten für weitere Implementierungen möglichst flexibel zu gestalten wurden Interfaces in Kombination mit Reflections verwendet. Konkret bedeutet dies, dass die neue Implementierungen von Protokollen, Knoten und Algorithmen in den entsprechenden Package erstellt werden müssen. Um korrekte Klassen zu erhalten welche den bestehenden Simulator ergänzen müssen die Methoden aus den dafür vorgesehenen Interfaces (siehe Abschnitt 4) überschrieben werden müssen.

Paket-Hierarchie

Unser Simulator ist in die folgenden Pakete unterteilt:

- **algorithm:** In diesem Paket finden sich die Routing-Algorithmen.
- **controller:** Sammlung der wichtigsten Klassen für die Simulationslogik.
- **data:** Datenstrukturen für die Kommunikation im Simulator.
- **event:** Diese Klassen ermöglichen den korrekten Ablauf der Kommunikation innerhalb des Netzwerks.
- **exception:** Behandlung von Ausnahmen bei der Verwendung von Reflections.
- **interfaces:** Beinhalten die Schnittstellen für die Erweiterbarkeit des Simulators.
- **model:** Verwaltet alle Typen von Knoten die vom Simulator verwendet werden können.
- **protocol:** Protokolle die für die Kommunikation verwendet werden können.
- **view:** Grafische Benutzeroberfläche.

Um eine klare Trennung der Klassen zu erreichen wurden diese in Pakete aufgeteilt. Die Aufspaltung wurde an das MVC-Pattern angelehnt. Das heißt, die Model - View - Controller Architektur

wurde als Basis für die Struktur verwendet. Ausführliche Erläuterungen, welche sich im jeweiligen Paket befinden, werden in den Folgekapiteln angeführt.

Interfaces

Interfaces dienen als Schnittstelle und können von Klassen verwendet werden. Wenn Klasse ein Interface implementiert wird gewährleistet, dass eine neue Klasse im Simulator alle benötigten Methoden implementiert und verwendet.

Model

Das Paket `model` enthält alle Klassen welche Entitäten innerhalb des Simulators repräsentieren. In unserem Fall sind dies die Implementierungen der Knoten (Nodes und Foes) welche für die Kommunikation im Netzwerk zuständig sind. Die Model-Klassen `Node`, `FoeCrowdsCOLLAB`, `FoeCrowdsCP`, `FoeCrowdsDOS` wurden hier für die Simulation einer Kommunikation des Crowds-Protokolls implementiert. Sollen weitere Knoten dem Simulator hinzugefügt werden müssen diese im Paket `model` liegen. Auch die bestehenden Klassen, welche auf die Kommunikation des Crowds-Protokolls angepasst sind, können um beliebige Funktionalitäten erweitert und die bestehenden Methoden bearbeitet werden.

Controller

Die Klassen im Paket `controller` enthalten die Logik für den Simulator. In unserem Fall sind dies die Klassen: `EventHandler`, `InputParser`, `LogHandler`, `NetworkGenerator` und der `Simulator` selbst.

Algorithm and Protocol

Klassen in den Pakets `algorithm` und `protocol` wären üblicherweise dem Paket `controller` unterzuordnen. Aber die Trennung von `algorithm` und `protocol` ist insofern von Bedeutung, da bei neuen Implementierungen von Protokollen und Algorithmen diese in den dazu vorgesehenen Pakets enthalten sein müssen. Somit gelten diese Pakets als "Sammlung" um sie in weiterer Folge dynamisch einzulesen.

Data

`Data` enthält Klassen welche die Struktur des Netzwerks abbilden. Diese sollten nach Möglichkeit nicht verändert werden. Die von uns vorgesehenen Infrastruktur und Logik im Bezug auf die Erweiterbarkeit ermöglichen eine Einbindung weiterer Protokolle und Algorithmen ohne in der Grundstruktur, eben dieses `data`-Paket, Anpassungen vorzunehmen. Die Struktur des Netzwerks ist demnach in der Klasse `Network` gespeichert. Die weiteren Klassen (`Paket`, `ReachableList`, `ReachableNodes`) dienen als Datenstrukturen um die Speicherung des Netzwerks auszulagern und die Kohäsion zwischen Klassen zu erhöhen.

Event

Das Paket `event` hält die Klassen die vom `EventHandler` zur Abarbeitung der Kommunikation benötigt werden. Die Einführung dieses Pakets diene hauptsächlich zur Verbesserung der Übersicht. Falls weitere Events hinzugefügt werden müssen, können diese problemlos in diesem Paket enthalten sein. Allerdings muss der `EventHandler` erweitert und adaptiert werden (mehr dazu siehe Abschnitt 4)

Exceptions

Eine Person die den Simulator anpasst hat die Möglichkeit weitere Algorithmen, Protokolle und Knotentypen zu erstellen. Zur Überprüfung der Korrektheit dieser neu hinzugefügten Klassen wurden dementsprechend **Exceptions** eingeführt, welche geworfen werden falls die Klassen nicht gefunden werden oder bei der Generierung, oder in der Netzwerkspezifikation, falsch eingegeben wurden.

View

Um die Grundzüge des MVC-Patterns beizubehalten beinhaltet das Paket **view** das Layout und Design sowie JavaFX-Controller. Man beachte, dass in diesem Paket Controller unüblich sind. Aber diese Controller haben die Aufgabe die Kommunikation zwischen User-Interface und der Logik zu ermöglichen. Die Controller übernehmen auch das Laden der Oberflächen sowie die Aktionen welche bei Klicks auf Buttons ausgeführt werden. Die Dateien mit den Endungen “fxml” stellen das Layout zur Verfügung und die CSS-Datei beinhaltet die Designspezifikationen.

Im folgenden Kapitel werden wir nun in der oben genannten Reihenfolge der Paket-Hierarchie auf die einzelnen Pakets und ihre Klassen eingehen.

Grundstruktur

Die Grundstruktur unseres Projektes wird nun an den drei wichtigsten Komponenten dargestellt. Diese sind **algorithm** (siehe Abbildung 4.1), **node** (siehe Abbildung 4.2) und **event** (siehe Abbildung 4.3).

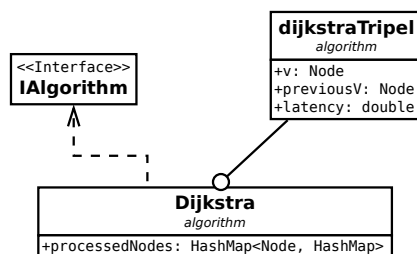


Abbildung 4.1: Grundstruktur für Algorithmen in unserem Simulator. Die Grafik wurde mit dem freien Tool DIA gezeichnet.

Auf die Darstellung der Controller-Klassen wurde bewusst verzichtet, da diese nicht (signifikant) zum besseren Verständnis beitragen würde. Wie in der Grafik illustriert, erkennt man, dass die Klasse **Node** die zentrale Entität ist. Dies ist darauf zurückzuführen, dass die Logik für die Kommunikation in der Klasse **Node** und ihren Subklassen enthalten ist. Auf die Einzelheiten gehen wir in Abschnitt 4 genauer ein. Die UML-Diagramme wurden unter GNU/Linux mit dem Tool *DIA* gezeichnet und als Vektorgrafiken eingebunden.

Interfaces

Interfaces stellen eine Sammlung von abstrakten Methoden zur Verfügung. Diese müssen von den Klassen welche die Interfaces implementieren überschrieben werden. Folgende Interfaces werden im Netzwerksimulator für die Implementierung des Crowds-Protokolls (2), des Dijkstra-Algorithmus (3) und den Knoten (4) verwendet.

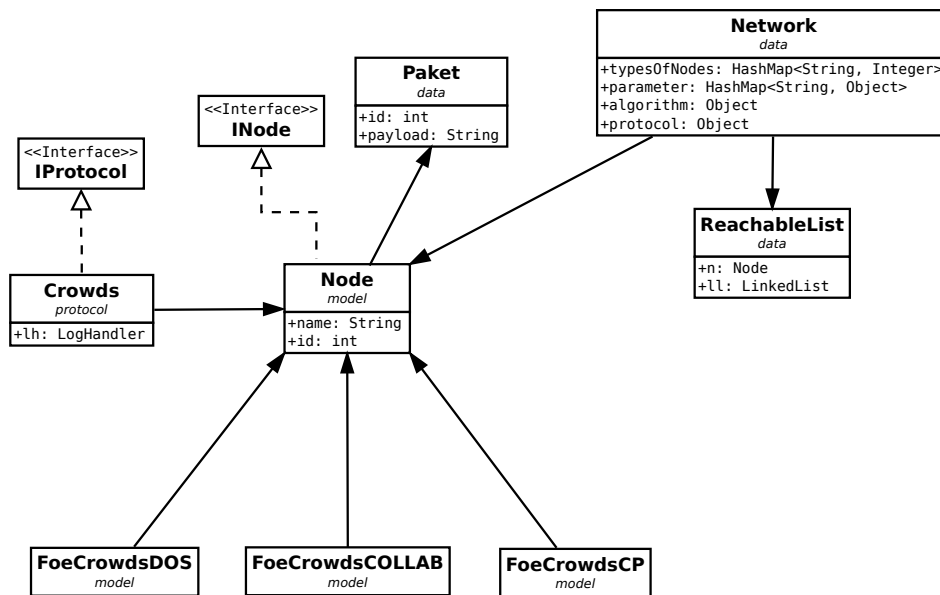


Abbildung 4.2: Grundstruktur für die Klasse Node die in unserem Simulator verschiedene Knoten implementiert. Die Grafik wurde mit dem freien Tool DIA gezeichnet.

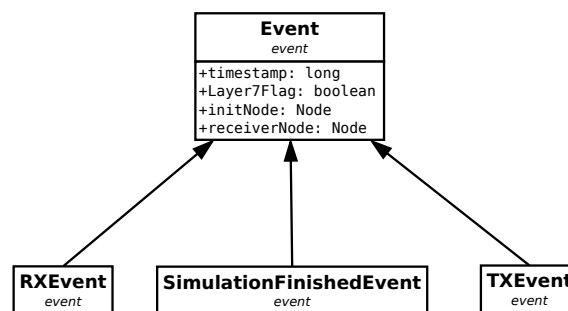


Abbildung 4.3: Grundstruktur für Events in unserem Simulator. Die Grafik wurde mit dem freien Tool DIA gezeichnet.

IAlgorithm

IAlgorithm, stellt das Interface mit den zu implementierenden Methoden zu Verfügung, um möglichst einfach, den Simulator um weitere Algorithmen, wie zum Beispiel den Algorithmus von *Floyd-Warshall*, zu erweitern [5].

```

public interface IAlgorithm
{
    public HashMap<Node, LinkedList<Node>> getPath(Node startNode);
}
  
```

Auf die vorhandene Implementierung der Methode **getPath** wird in 4 noch eingegangen.

INode

Das Interface **INode** deklariert abstrakte Methoden die für die Funktionalität der Übertragung zwischen Knoten im Netzwerk benötigt werden. Jede Klasse welche einen **Node** repräsentiert

muss dieses Interface implementieren und somit die untenstehenden Methoden überschreiben um die Kommunikation zu ermöglichen.

```
public interface INode
{
    public Event receive(Node initNode, Node nextNode, Paket p);
    public Event receiveLayer3(Node initNode, Node nextNode, Node receiver, Paket p);
    public RXEvent transmit(Node initNode, Node nextNode, Paket p);
    public RXEvent transmitLayer3 (Node initNode, Node nextNode, Paket p);
    public TXEvent startCommunication();
}
```

Eine detaillierte Erklärung der Methoden ist in Abschnitt 4 angeführt.

IProtocol

IProtocol wurde eingeführt um die Implementierung von Protokollen zu vereinheitlichen. Um auch bei dem verwendeten Protokoll Erweiterbarkeit maximale zu erreichen wurden neben zwei `execute`-Methoden (`executeTx`, `executeRX`) auch die Methoden `executePreSimulation`, `executeFinished` und `executePostSimulation` zur Implementierung zu Verfügung gestellt. `executeTx` wird verwendet um protokoll-spezifische Aktionen bei einem Transmit-Event (siehe 4) auszuführen. Als Gegenstück wird `executeRx` aufgerufen um beim Empfang von Paketen zu reagieren. Die Methode `executeFinished` wird nach einem `executeTx` oder `executeRx` ausgeführt. Um vor der Simulation Konfigurationen oder andere Methoden zu ermöglichen wird `executePreSimulation` verwendet und um nach einer Simulation definierte Aktionen durchführen zu lassen kann die Methode `executePostSimulation` mit Funktionalität versehen werden. Jeder dieser Möglichkeiten steht eine beliebige Anzahl von Übergabeparametern zu Verfügung. Das hat den Vorteil, dass man bei einer eigenen Implementierung nicht auf eine fixe Anzahl von Parameter eingeschränkt ist.

```
public interface IProtocol
{
    public void executeTX(Object... object);
    public void executeRX(Object... object);
    public void executePreSimulation(Object... object);
    public void executeFinished(Object... object);
    public void executePostSimulation(Object... object);
}
```

Die Übergabeparameter vom Typ `Object` können beliebig Objekte übernehmen und der Entwickler eines neuen Protokolls kann somit beliebig viele Parameter übergeben um sie im spezifizierten Protokoll zu verwenden. Sollte keine Funktionalität benötigt werden kann die Methode leer gelassen werden.

Da wir nun einen Überblick über die Interfaces bekommen haben werden wir die Details näher anführen. In diesem Zuge werden wir, im Sinne der Übersichtlichkeit, zuerst das Zusammenspiel für Erweiterbarkeit zwischen Interfaces und Reflections erläutern.

Erweiterbarkeit

Um ein hohes Maß an Erweiterbarkeit zu erreichen, können beliebige Protokolle und Algorithmen hinzugefügt werden. Die bestehenden Repräsentationen können als Referenz für weitere

Implementierungen herangezogen werden. Durch die Verwendung von Interfaces ist es möglich neue Klassen, welche die Eigenschaften der Kommunikation beeinflussen, sehr einfach hinzuzufügen. Die neuen Klassen müssen das dafür vorgesehene Interface implementieren und die benötigten Methoden überschreiben. Reflections ermöglichen es, die Klassen zur Laufzeit zu bestimmen, zu überprüfen und dynamisch die benötigten Objekte zu erzeugen. Weiteres können, neben Protokollen und Algorithmen, zusätzliche Knotentypen, für die Umsetzung von weiteren Angreifern beziehungsweise Angreifertypen, hinzugefügt werden. Wieder liefern Interfaces die nötige Strukturdefinition um eine Einbindung mit geringem Aufwand für den/die AnwenderIn zu ermöglichen. Eine Konfigurationsdatei ermöglicht es, Parameter und Einstellungsmöglichkeiten an den Simulator zu übergeben. Die Mitgabe von Parametern kann auch im Generator (siehe Abschnitt 4) erfolgen. Der Zugriff auf Variablen innerhalb der Kommunikation wird jederzeit ermöglicht. Die folgende Auflistung zeigt eine Übersicht über die Elemente des Netzwerksimulators die Einfluss auf die auf die Erweiterbarkeit haben:

- Protokolle für die Kommunikation und protokollspezifische Aktionen.
- Algorithmen für die “Wegfindung” durch das Netzwerk.
- Knoten welche das Verhalten (Senden, Empfangen, etc.) eines Rechners simulieren.
- Parameter welche die spezifizierten Eigenschaften verwenden können.

Die Konventionen welche die Interfaces vorgebe, müssen strikt eingehalten werden, da der gesamte Netzwerksimulator auf diese Spezifikationen aufbaut. Werden diese Konventionen eingehalten führt der Simulator den Ablauf der Kommunikation mit den neu spezifizierten “Elementen” durch.

Ein Beispiel für eine Konfigurationsdatei findet sich in Listing 4.1. Wenn ein Netzwerk automatisch generiert werden soll, so können die Parameter, anstelle der Konfigurationsdatei, in einer Graphischen Benutzeroberfläche übergeben werden (siehe 6.2). Mit diesen Einstellungen kann der Algorithmus und das zu verwendende Protokoll gewählt werden. Dieses wird anschließend auf Korrektheit und Existenz überprüft. In weiterer Folge abhängig vom Typ eine Instanz dieser Klasse erzeugt. Der Algorithmus und das Protokoll müssen im zugehörigen Paket (siehe 4) vorhanden werden. Die Objekte der Protokoll- und Algorithmus-Klasse sind, für den Zugriff innerhalb des Simulators, in der Klasse `Network` deklariert:

```
public class Network
{
    public static Object algorithm;
    public static Object protocol;
    ...
}
```

Die Methoden der Instanzen werden vom Netzwerksimulator im späteren Verlauf zur Kommunikation verwendet. Beispiele dafür wären die Wegfindung durch das Netzwerk und die Weiterleitung der Pakete zum Empfänger.

Das Erzeugen und Initialisieren der “richtigen” (die Klassennamen welche in der Konfiguration angegeben wurden) Objekte wurde mittels Reflections und Interfaces ermöglicht. Im folgenden Codeausschnitt zeigen wir wie die Erzeugung des Objektes mittels Reflections erfolgt. Als Bei-

spiel, zeigen wir dies für den spezifizierten Algorithmus. Analog dazu geht die Erzeugung vom Protokoll und Knoten von statten.

```

IAlgorithm ialgorithm = createObjectAlgorithm(classType);
if (ialgorithm == null)
{
    throw new AlgorithmNotFoundException();
}
else
{
    Network.setAlgorithm(ialgorithm);
}

```

In der Klasse `InputParser` (siehe 4) wird aus der Konfigurationsdatei (oder falls die Oberfläche der Netzwerkgenerierung verwendet wird, in der Klasse `NetworkGenerator` (zu finden unter 4)) der Name der Klasse ausgelesen. Dieser befindet sich als Folge in `classType`. Die Methode `createObjectAlgorithm` gibt ein Objekt vom Typ des Interfaces zurück. Es wird eine `Exception` geworfen falls die Klasse nicht gefunden wird oder bei der Erstellung der Instanz ein Fehler auftritt. Der folgende Programmausschnitt zeigt die Methode der Erstellung eines Algorithmus:

```

private IAlgorithm createObjectAlgorithm(String classType)
{
    IAlgorithm castToIAlgorithm = null;
    try
    {
        Class classDefinition = Class.forName("algorithm." + classType);
        castToIAlgorithm = (IAlgorithm) classDefinition.newInstance();
    }
    catch (InstantiationException e)
    {
        if(debug) System.out.println(e);
    }
    catch (IllegalAccessException e)
    {
        if(debug) System.out.println(e);
    }
    catch (ClassNotFoundException e)
    {
        if(debug) System.out.println(e);
    }
    return castToIAlgorithm;
}

```

Die Methode übernimmt als Übergabeparameter den Namen des Algorithmus (und de facto auch der Klasse, als Beispiel "Dijkstra"). Mit diesem Übergabeparameter wird die Klasse im dem Paket gesucht. Falls diese vorhanden ist, wird der Variable `classDefinition` diese Klasse zugewiesen. Die Instanz des Interfaces (hier `castToIAlgorithm`) wird im nächsten Schritt mit der einer neuen Instanz, die auf `IAlgorithm` gecastet wird, initialisiert. Was auffällt ist, dass der Rückgabewert der Methode das Interface `IAlgorithm` ist. Dies ist notwendig, da wir nicht ein Objekt einer speziellen Klasse zurückgeben wollen, sondern eben ein Objekt, welches das zugehörige Interface implementiert. Analog werden auch alle Knoten und das Protokolle erzeugt.

Konfigurationsdatei und Netzwerkparameter

Um die Auswahl des zu verwendenden Protokolls zu erleichtern, kann diese direkt vor dem Beginn der Simulation in einer Konfigurationsdatei festgelegt werden. Dies kann bei der Generierung des Netzwerks über ein Parameterfeld in der graphischen Benutzeroberfläche geschehen. Bei Verwendung einer Input-Liste, welche die Graph-Spezifikation im Sinne einer Kantenliste darstellt, müssen diese Details am Beginn der Simulation in der Parameterliste angegeben werden.

Neben dem Protokoll welches in der Simulation verwendet werden soll, muss auch der Algorithmus für das Routing angegeben werden. Dies erfolgt, ebenso wie bei Protokollen, entweder über die Konfigurationsdatei oder die Parameterliste. Am Ende der Liste können zusätzliche Parameter für das Protokoll eingegeben werden, welche durch die Simulator-Klasse in Form einer `HashMap` global verfügbar gemacht werden. Wie bereits erwähnt, wird auch die Möglichkeit geboten verschiedenste Parameter im Netzwerksimulator zu Verfügung zu stellen. Wieder erfolgt die Eingabe auf die selbe Weise wie zuvor. Die Parameter werden nach dem Einlesen in einer `HashMap` mit einem `String` als Schlüssel und einem `Object` als Wert gespeichert. Wir entschieden uns für eine `HashMap` da diese eine schnelle Suche bereitstellt und die Inhalte bereits als “Key - Value - Paare” realisiert sind.

```
public static HashMap <String, Object> parameter;
```

Der Zugriff auf Parameter kann aus jedem Protokoll, Algorithmus oder Knoten erfolgen. Spätestens zum Zeitpunkt wenn die Simulationsoberfläche (6.3) aufscheint ist das Netzwerk vollständig erstellt. Für Text sollte man den Typ `String` verwenden und für Zahlen den Typ `Float`. Der Wert ist mittels den Schlüssel der `HashMap`, einem eindeutigen String, zugreifbar. Der Zugriff erfolgt über die Methode `getParameter`, welche in der `HashMap` nach dem entsprechenden String sucht.

Es wird ein Objekt, abhängig vom Typ welcher dem Schlüssel zugewiesen, wurde zurückgegeben. Als Beispiel die Wahrscheinlichkeit im Crowds-Protokoll (2), dass ein Paket (4) weitergeleitet wird. Als Schlüssel wäre dies der `String` “pf” und als Wert ein `Float`.

Model

Im Paket `Model` befinden sich die Klassen welche Objekte aus dem Netzwerk bereitstellen. Auch zukünftige Implementierungen von weiteren Knotentypen ist dies das zu verwendende Paket. Als Beispiel, ein neuer Knoten um weitere Angriffsszenarien abzudecken.

Node

Die folgenden Implementierungsdetails befassen sich mit dem Crowds-Protokoll, sind aber im Weiteren durch Adaptierungen auf weitere Protokolle anwendbar. Die Klasse `Node` bietet die *Basisimplementierung* eines Knotens, welcher einen “Benutzer”, eine “Benutzerin” oder eine “Instanz” eines Clients für ein Protokoll darstellt. `Node` implementiert das Interface `INode`, jede weitere Implementierung erbt die Funktionalitäten von dieser Klasse. Da ein Knoten eine Instanz im Netzwerk darstellt und somit auch der initiale Sender (Initiator) sein kann, muss es auch die Möglichkeit geben eine Verbindung zu starten bzw. wie im Crowds-Protokoll festgelegt zuerst an einen beliebigen Empfänger zu senden. Folgende Methode übernimmt diese Aufgabe:

```

public TXEvent startCommunication()
{
    Network network = Network.getInstance();
    int n = (int) (getSecureRandomNumber() * network.getAllNodes().size());
    Node receiver = network.getAllNodes().get(n);
    TXEvent startEvent = new TXEvent(this, receiver);
    return startEvent;
}

```

Diese Methode wird zu Beginn jeder Übertragung aufgerufen. Zu Beginn wird eine Instanz der Klasse `Network`, welche mittels Singleton-Pattern implementiert wurde, referenziert. Das Singleton-Pattern bezeichnet ein Muster zur Objekterstellung, bei dem immer nur eine Instanz vorhanden sein kann. Diese Instanz ermöglicht es Zugriff auf die Anzahl der Knoten und infolge auf den Knoten selbst zu erhalten, welcher als der ausgesuchte Empfänger in einem `TXEvent` an den `EventHandler` zurückgegeben wird.

Weiteres spielen zwei Methoden bei der Kommunikation eine besondere Rolle. Einerseits `transmit` und andererseits `receive`. Wie der Name der Methoden schon vermuten lässt enthält `transmit` die Funktionalität zum Senden und `receive` selbige zum Empfangen. Anzumerken ist, dass die Layer 3 und Layer 7 Implementierungen des OSI Schichtenmodells (siehe 2) in die Methoden `transmitLayer3` und `receiveLayer3` ausgelagert sind. Je nach `Event` und “Layer-Flag” werden die Methoden aus dem `EventHandler` aufgerufen. Im folgenden Codebeispiel gehen wir schrittweise auf die Implementierung der `receive`-Funktion ein:

```

public Event receive(Node sNode, Node rNode, Paket p)
{
    /** sNode = Startnode, rNode = ReceiverNode */
    EventHandler ev = EventHandler.getInstance(sNode,rNode,p);
    Node endNode = ev.getEndNode();

    if(endNode.equals(rNode))
    {
        SimulationFinishedEvent eFinish = new SimulationFinishedEvent(rNode, endNode);
        eFinish.setLayer7Flag(true);
        return (Event)eFinish;
    }
}

```

Die `receive`-Methode bekommt als Parameter den Startknoten, den Empfängerknoten und das zu sendende Paket übergeben. Mit einer Instanz des `EventHandlers` wird der Endknoten (zum Beispiel ein Server) geholt. Falls beim Übertragen des Paktes der Empfänger der Endknoten ist wird ein `SimulationFinishedEvent` (“Layer 7”) erstellt und an den `EventHandler` zurückgegeben.

```

double pForward = Double.parseDouble(Network.getParameter("pf").toString());
double pRandom = getSecureRandomNumber();

```

In diesem Abschnitt der `Node`-Klasse werden zwei `Double`-Werte initialisiert. Einerseits wird die “Probability-to-Forward” (`pf`, ein Crowds-spezifischer Parameter) festgelegt und andererseits ein Zufallswert. Beide werden im nächsten Codeabschnitt verwendet um den Folgeknoten festzulegen.

```

if(pForward > pRandom)
{

```

```

System.out.println("L7 | --- Get random node for next transmit --- ");
int n = (int) (getSecureRandomNumber() * Network.getAllNodes().size());
Node receiver = Network.getAllNodes().get(n);
TXEvent txe = new TXEvent(rNode, receiver);
txe.setLayer7Flag(true);
System.out.println("--- " + rNode + " -> " + receiver );
return (Event)txe;
}

```

Die beiden vorher festgelegten Werte werden verglichen. Ist die Wahrscheinlichkeit mit der im Netzwerk weitergeleitet wird größer als der Zufallswert p_{Random} , so erfolgt die Übermittlung an einen beliebigen Knoten im Netzwerk.

```

else
{
    System.out.println("L7 | --- End with transmit to final receiver --- ");
    TXEvent txe = new TXEvent(rNode, endNode);
    txe.setLayer7Flag(true);
    return (Event)txe;
}
}

```

Ist dies nicht der Fall und die Zufallszahl ist größer als die “Weiterleitungswahrscheinlichkeit”, wird das Paket zum Endknoten übermittelt. Hierbei handelt es sich jeweils um Layer 7 Events, das bedeutet, dass der tatsächliche Weg durch das Netzwerk erst noch berechnet werden muss (dies geschieht im `EventHandler`). Die weiteren Methoden (`receiveLayer3`, `transmit` und `transmitLayer3`) sind zwar essentiell, jedoch enthalten diese weniger Funktionalität, sodass wir nicht näher darauf eingehen werden.

In den weiteren Unterkapiteln werden wir nun auf mehrere Spezialfälle von Node-Implementierungen angeführt. Da im Zuge der Crowds-Implementierung und da Angriffsszenarien auf Knotenebene stattfinden, mussten neue Knoten-Klassen spezifiziert werden, welche Funktionen für Angriffe bereitstellen. Ziel dieser Implementierungen ist es die Kommunikation des Protokolls mittels den Knotentypen “Change-Paket” (`FoeCrowdsCP`), “Denial of Service” (`FoeCrowdsDOS`) und mit mehreren “kollaborierenden Knoten” (`FoeCrowdsCollab`) anzugreifen und die Ergebnisse zu analysieren.

FoeCrowdsCP

Die Klasse `FoeCrowdsCP` hat die Aufgabe ein empfangenes Paket zu ändern und das geänderte Paket weiterzuleiten (aktiver Angriff). Dies simuliert, zum Beispiel, eine Änderung der Kontodaten in einer Übertragung im Internet. `FoeCrowdsCP` erbt die Standardimplementierung eines Crowds-Knoten (welcher in unserer Implementierung die Klasse `Node` ist) und benötigt somit nur ein Überschreiben der `transmitLayer3`-Methode. Exemplarisch wird bei jeder Beteiligung der Inhalt des Pakets auf “changed!” gesetzt.

```

public class FoeCrowdsCP extends Node{
{
    ...
    public RXEvent transmitLayer3(Node initNode, Node nextNode, Paket p)
    {
        p.setPayload("changed!");
        nextNode.setP(p);
        RXEvent rxe = new RXEvent(initNode, nextNode);
        return rxe;
    }
}
}

```



```

    }
    ...
}

```

In dieser Methode wird das Paket verändert und im neuen Event an die Folgeknoten weitergegeben. Die weitere Kommunikation erfolgt nun mit den geänderten Paket.

FoeCrowdsDOS

Ein **FoeCrowdsDOS** ist ein weiteres Beispiel für ein Denial-of-Service-Angriffsszenario im Crowds-Protokoll. Hier wird die weitere Kommunikation mit den Folgeknoten unterbunden. Diese Repräsentation eines Knoten soll einen Ausfall im Netzwerk oder eine gewollte Verweigerung der Weiterleitung simulieren. Hierzu werden die Methoden `receive`, `transmitLayer3` und `startCommunication` überschrieben. Es wird die Simulation beendet und kein Event zurückgegeben. Somit wird keine weitere Kommunikation ermöglicht, empfängt ein DoS-Knoten ein Paket so wird die Verbindung beendet.

FoeCrowdsCOLLAB

Unsere letzte Implementierung eines Angriffs, auf das von uns gewählte Protokoll, bei dem die Knoten untereinander zusammenarbeiten nennt sich **FoeCrowdsCOLLAB**. Die “Collaborating Foes” arbeiten zusammen um den initialen Sender der Kommunikation auszuforschen. Die von uns durchgeführte empirische Untersuchung beruht darauf, dass die kollaborierenden Angreifer die IDs der Sender, von denen sie Pakete empfangen, in einer Liste speichern. Diese Liste wird in der `Protokoll`-Klasse gespeichert und am Ende der Kommunikation überprüft. Der genaue Ablauf wird im Abschnitt 2 beschrieben.

Controller

Neben der in den Knotenklassen enthaltenen Protokoll-Logik befinden sich wichtige Funktionalitäten, welche den Simulationsablauf regeln, im Paket **Controller**. Die Aufgaben der Klassen reichen vom Einlesen des Netzwerks (`InputParser`) und der Netzwerkgenerierung (`NetworkGenerator`) bis zum Abarbeiten der Kommunikation (`EventHandler`) und deren Protokollierung (`LogHandler`). Da der `EventHandler` der zentrale Drehpunkt der Kommunikation ist, werden wir zuerst auf seine Implementierung eingehen.

EventHandler

Der `EventHandler` verwaltet Events, die von Nachrichten erzeugt werden, mittels einer `PriorityQueue`. Diese `Queue` ist eine von Java bereitgestellte Implementierung eines Min-Heaps. Die Events werden sequentiell abgearbeitet und die entsprechenden Aktionen ausgeführt. Es werden weitere Events initiiert um mit der Kommunikation fortzufahren, oder um diese zu beenden.

Hinsichtlich der Trennung von Vermittlungs- und Anwendungsschicht wurde der Ablauf unserer Implementierung wie folgt umgesetzt: Der `EventHandler` erhält von einem Knoten, welcher die Übertragung (bzw. die Kommunikation) initiiert, ein Event für das Senden einer Nachricht über die Anwendungsschicht. Mittels des Dijkstra-Algorithmus wird der kürzeste Weg vom Sender zum Empfänger ermittelt. Die “logische” Übertragung (der Anwendungsschicht) wird in Folge

durch die entsprechenden “physikalischen” Übertragungen zwischen den Knoten im Netzwerk (der Vermittlungsschicht) ersetzt. Am Ende einer Übertragung wird als Resultat das Paket auf Anwendungsschicht empfangen. Die Anzahl der Nachrichten auf der Vermittlungsschicht wird hierbei vom kürzesten Weg bestimmt. Falls diese bereits kürzesten Weg verbunden sind kann die Übertragung direkt zwischen den Knoten erfolgen.

```
private PriorityQueue<Event> queue;
private IProtocol protocol;

public boolean simulate(Network n)
{
    protocol.executePreSimulation(null);
    ...
    /* Start of communication. Add the start-event (via the startNode) */

    while (!this.queue.isEmpty())
    {
        e = this.queue.poll();
        if (e instanceof SimulationFinishedEvent)
        {
            executeEvent(e);
            return false;
        }
        else
        {
            executeEvent(e);
        }
    }
}
protocol.executePostSimulation(null);
```

Um zwischen den Events und den Übertragungsschichten zu unterscheiden wurde die Methode `executeEvent` implementiert. Die folgenden Codesegmente werden die einzelnen Typen beschreiben und welche Aktion durchgeführt wird. Jedes der Segmente befindet sich in der Methode `executeEvent` und diese ist wie folgt ausgeführt:

```
...
private LinkedList<Node> listForCommunication;
private IAlgorithm algorithm;
...

private void executeEvent(Event e)
{
    Node sNode = ((Event) e).getInitNode();
    Node rNode = ((Event) e).getreceiverNode();
```

Die Methode nimmt ein `Event` entgegen und klassifiziert durch Abarbeitung der festgelegten Spezifikationen wird je nach Typ die dazugehörige Aktion durchgeführt.

```
/* Finishing the communication */
if (e instanceof SimulationFinishedEvent)
{
    rNode.receive(sNode,rNode, sNode.getP());
    protocol.executeFinished(null);
}
```

Wird ein Event vom Typ `SimulationFinishedEvent` übergeben so wird die `receive` Methode beim Empfänger aufgerufen und in Folge kein weiteres Event aus der Queue geholt und in

Folge der Simulator beendet. Wird jedoch ein Event vom Typ `RXEvent` übergeben, so wird der folgende Zweig der Bedingung betreten und eine Unterscheidung zwischen Layer 3 und Layer 7 muss getroffen werden.

```

/* receive */
else if (e instanceof RXEvent)
{
    if(e.isLayer7Flag())
    {
        Event ev = rNode.receive(sNode, rNode, sNode.getP());
        this.addEvent(ev);
    }
    else
    {
        if(listForCommunication.size() > 0)
        {
            Event ev = rNode.receiveLayer3(sNode, rNode, listForCommunication.pollFirst(), sNode.getP());
            this.addEvent(ev);
        }
        else
        {
            Event ev = rNode.receiveLayer3(sNode, rNode, null, sNode.getP());
            this.addEvent(ev);
        }
    }
}
protocol.executeRX(null);

```

Wird ein Layer 7 Event ermittelt, so wird die `receive` Methode des Empfängers aufgerufen und das Event welches zurückgegeben wird in die Queue hinzugefügt. Falls ein Layer 3 Event auftritt, wird der nächste Empfänger aus der Liste gewählt, die mittels Dijkstra-Algorithmus erstellt wurde. Die Methode `receiveLayer3` vom Empfängerknoten wird aufgerufen. Je nach Implementierung im Knoten wird ein Event zurückgegeben und zur Queue hinzugefügt. Um Erweiterbarkeit zu unterstützen wird nach der Abarbeitung eines `RXEvents` die protokollspezifische Methode `protocol.executeRX(null)` aufgerufen, wobei je nach Implementierung der Methode in der Protokollklasse beliebig viele Parameter übergeben werden können. Natürlich gibt es auch einen Entscheidungszweig für den Fall eines `TXEvent`.

```

else if (e instanceof TXEvent)
{
    if (e.isLayer7Flag())
    {
        try
        {
            listForCommunication = algorithm.getPath(sNode, rNode);
            if(listForCommunication == null)
            {
                lh.appendData("Node cannot be reached");
                System.out.println("Node cannot be reached");
            }
            else
            {
                System.out.println(listForCommunication.toString());
                if(listForCommunication.size() > 1)
                {
                    Node firstNode = listForCommunication.pollFirst();
                    Node nodeToSendTo = listForCommunication.pollFirst();
                    firstNode.setP(sNode.getP());
                    TXEvent eFirst = new TXEvent(firstNode, nodeToSendTo);
                    eFirst.setLayer7Flag(false); //Layer 3
                }
            }
        }
        catch (Exception ex)
        {
            // ...
        }
    }
}

```

```

        this.addEvent(eFirst);
    }
    else
    {
        /* Sending to the same node */
        RXEvent eFirst = new RXEvent(sNode, rNode);
        eFirst.setLayer7Flag(true); //Layer 7
        this.addEvent(eFirst);
    }
}

}
catch (Exception e1)
{
    if(debug) e1.printStackTrace();
}
}

```

Wieder wird die Unterscheidung zwischen Layer 7 und Layer 3 getroffen. Ist ersteres der Fall, wird der Pfad mittels Dijkstra-Algorithmus ermittelt und eine Liste (`listForCommunication`) mit Knoten befüllt. Von Liste der Knoten welche an der Kommunikation beteiligt sind, wird bei jeder Übertragung das letzte Element ausgewählt und entfernt. Ist die Größe der Liste kleiner gleich 1 so wird an den selben Knoten geschickt. Falls das Event ein Layer 3 Event ist so wird die `transmitLayer3` Methode aufgerufen und ein weiteres Layer 3-Event zur Queue hinzugefügt.

```

/* Layer3 just execute */
else
{
    try
    {
        RXEvent eLayer3 = rNode.transmitLayer3(sNode, rNode, sNode.getP());
        eLayer3.setLayer7Flag(false);
        this.addEvent(eLayer3);
    }
    catch (Exception e1)
    {
        if(debug) e1.printStackTrace();
    }
}
protocol.executeTX(null);

```

Wieder folgt nach Ende der Abarbeitung der Type von Event eine protokollspezifische Methode `protocol.executeTX(null)`.

InputParser

Der Inputparser liest eine Text-Datei ein, aus der ein Netzwerk erzeugt werden soll. Die Eingabedatei besteht aus 3 Teilen und wird in Listing 4.1 dargestellt.

```

# Dies ist ein Kommentar
Protocol: Crowds
Algorithm: Dijkstra
Parameter
pf 0.7
-
0 Node
1 Node
2 Node
3 Node

```

```

-
0 1 76
0 3 36
1 2 70
1 3 12
2 3 21

```

Src. 4.1: Beispiel einer Eingabedatei für Netzwerkgenerierung

In der Datei können Zeilen mittels `#` auskommentiert werden. Der Inputparser liest die verschiedenen Teile der Eingabedatei separat ein und liest solange, bis die Zeilen zu Ende sind oder eine Zeile zu Beginn ein `-` enthält. Für den ersten Teil der Eingabedatei in der die Parameter für die Simulation spezifiziert werden, wird hier exemplarisch das Parsen der Eingabedatei gezeigt.

```

if(line.startsWith("Algorithm"))
{
    buffer = line.split(":");
    String classType = buffer[1].trim();
    IAlgorithm ialgorithm = createObjectAlgorithm(classType);
    if (ialgorithm == null)
    {
        throw new AlgorithmNotFoundException();
    }
    else
    {
        Network.setAlgorithm(ialgorithm);
    }
}

```

Da Reflections verwendet wurden, ist es wichtig, dass der Name der als Parameter angegeben wird mit dem Klassennamen übereinstimmt. Ansonsten wird eine `AlgorithmNotFoundException` geworfen. Der Aufbau für das Parsen des Protokolls ist gleich dem des Algorithmus. Einziger Unterschied in der Funktionsweise ist, dass eine `ProtocolNotFoundException` geworfen wird. Anschließend wird die Liste der Parameter für das Protokoll eingelesen. Dies erfolgt folgendermaßen:

```

else if(line.startsWith("Parameter"))
{
    line = reader.readLine();

    while(!line.startsWith("-")) // Do until the end of section
    {

        buffer = line.split(" ");
        name = buffer[0].trim();
        String stringValue = buffer[1];

        try
        {
            float value = Float.parseFloat(stringValue);
            Network.addParameter(name, value);
        }
        catch (Exception e)
        {
            String value = stringValue;
            Network.addParameter(name, value);
            if(debug) e.printStackTrace();
        }
    }
}

```

```

    }
    line = reader.readLine();
}

```

Das Parsen der Parameter beginnt sobald eine Zeile mit dem Text “Parameter” gelesen wird. Die einzelnen Zeilen werden nun in String-Arrays umgewandelt, wobei zur Trennung einzelne Leerzeichen verwendet werden. An der ersten Stelle im Array ist der Name des Parameters (welcher als Schlüssel verwendet wird), an zweiter Stelle im Array folgt der Wert des Parameters. Wenn der Wert nicht zu einer Gleitkommazahl umgewandelt werden kann, wird eine **Exception** abgefangen und dieser als **String** behandelt. Nun werden die Parameter eingelesen bis das Ende des ersten Eingabeblocks durch ein “-” beendet wird.

Nachdem die Parameter eingelesen wurden folgt das Parsen der einzelnen Knoten die im Netzwerk erzeugt werden. Pro Zeile wird ein Knoten spezifiziert, indem zunächst seine eindeutige Identifikationsnummer angegeben wird und anschließend der Knotentyp. Der Knoten wird nun wiederum durch Reflections erzeugt, weshalb der Name in der Eingabedatei mit dem Namen der Klasse übereinstimmen muss. Folgender Codeteil zeigt das Parsen der Knoten:

```

line = reader.readLine();

while (line != null && !line.startsWith("-"))
{
    if(!line.startsWith("#"))
    {
        buffer = line.split(" ");
        id = Integer.parseInt(buffer[0]);
        type = "model." + buffer[1];
        name = buffer[1];

        INode node = createObjectNode(type);
        if (node == null)
        {
            throw new NodeTypeNotFoundException();
        }

        node.setId(id);
        node.setName(name);

        allNodes.add(id, (Node) node);

        /* Creates the reachable for every node */
        ReachableList rl = new ReachableList(allNodes.get(id));
        nodesToReach.add(rl);
    }
    line = reader.readLine();
}

```

Auch im zweiten Teil der Liste werden Zeilen die mit einer # beginnen als Kommentare gewertet und übersprungen. Wird eine Zeile eingelesen, erfolgt wiederum die Erzeugung eines Arrays indem die Zeile anhand der Leerzeichen geteilt wird. Mit dem zweiten Parameter wird nun ein neuer Knoten generiert oder eine **Exception** geworfen, falls dieser nicht erzeugt werden kann. In diesem Fall wird eine **NodeTypeNotFoundException** geworfen und der Knoten ist nicht im Paket **model** vorhanden. Als “Namespace” wird die Zeichenkette “model.” zum Beginn des Knotentyps hinzugefügt. Der Knoten erhält nun die – als erster Parameter – angegebene Identifikationsnummer und als Name wird der Knotentyp eingetragen. Jeder erzeugte Knoten wird nun der Liste

`allNodes` hinzugefügt und eine `ReachableList` für den Knoten erzeugt. Wenn der Parser eine Zeile die mit einem “-” beginnt einliest ist der zweite Teil der Liste beendet und die Liste der Verbindungen zwischen den Knoten wird eingelesen.

Wenn die Parameter spezifiziert und alle benötigten Knoten erzeugt wurden, werden vom Parser die Verbindungen eingelesen und erzeugt. Pro Zeile kann eine Verbindung angegeben werden. Eine solche Verbindung besteht aus 3 Werten die übergeben werden müssen. Die ersten beiden Werte geben die Identifikationsnummern der Knoten an und der dritte Wert gibt die Latenz zwischen den beiden Knoten an. Untenstehend ist der Code für das Parsen der Verbindungen zu finden:

```
try
{
    nodesToReach.get(id).addReachable(allNodes.get(to), latency);
}
catch (Exception e)
{
    if (debug) e.printStackTrace();
    return null;
}

try
{
    nodesToReach.get(to).addReachable(allNodes.get(id), latency);
}
catch (Exception e)
{
    if (debug) e.printStackTrace();
    return null;
}
}
```

Nun wird in die `ReachableList` bei beiden Knoten die Verbindung eingetragen. Damit jeder Knoten nur über eine `ReachableList` verfügt werden diese schon beim Einlesen der Knoten erzeugt und nicht erst in diesem Schritt, da hier sonst eine getrennte Überprüfung stattfinden müsste. Zudem werden noch `Exceptions` des Zahlenformats abgefangen, da die ersten beiden Werte vom Typ `Integer` sein müssen und der dritte Wert eine Gleitkommazahl sein muss. Außerdem wird noch eine `IndexOutOfBoundsException` für das erzeugte Array abgefangen. Weitere `Exceptions` werden in der Konsole ausgegeben, wenn `debug = true` gesetzt ist. Wenn der Parser keine Zeilen mehr einliest wird das Parsen beendet. Ist der Vorgang erfolgreich, werden die erzeugten `Parameter`, `Nodes` und `ReachableLists` im Netzwerk gespeichert. Der Parser gibt als Abschluss die Instanz des Netzwerks zurück.

LogHandler

Die Klasse `LogHandler` aus dem Paket `controller` wird für die Erzeugung von *csv-Dateien* (Comma-separated-values) benötigt. Diese Dateien können in einer Tabellenkalkulationssoftware dargestellt werden und enthalten alle wichtigen Informationen zu einem Simulationsdurchlauf. Die Datei gibt Auskunft über den Layer auf dem die Kommunikation stattgefunden hat, den Absender, ob das Datenpaket versendet oder empfangen wurde und den Empfänger. Zudem ist für jede Aktion die aktuelle Zeit in Form eines Unix-Timestamps enthalten. Am Ende der Datei wird die gesamte Simulationszeit des Systems angezeigt. Die Dateien werden im Unterordner “Logs”, im Verzeichnis in dem die Sourcecode-Dateien vorhanden sind `src/Logs`, gespeichert.

Der `LogHandler` wird mittels “Singleton-Pattern” erzeugt, damit sichergestellt wird, dass immer nur eine Instanz des `LogHandlers` existiert. Die “Logs” werden nun mittels `FileWriter` geschrieben, oder bei Fehler eine `FileNotFoundException` geworfen.

```
/* Instantiate the FileWriter for logging */
public void writeHeader() throws IOException
{
    try
    {
        writer = new FileWriter(csvLogName+logcounter+".csv");
    }
    catch (FileNotFoundException e1)
    {
        new File(path + "/Logs").mkdirs();
        writer = new FileWriter(csvLogName+logcounter+".csv");
    }
    String str = "Layer;sender;action;receiver;paket text;timestamp\n";
    appendData(str);
}
```

Die einzelnen Events werden als Zeile nacheinander in einer *csv-Datei* hinzugefügt. Hierfür wird zunächst die Methode `formattingLogOutput` aufgerufen. Diese formatiert die Zeile die geschrieben werden soll in ein csv-konformes Format. Nun wird diese Zeile mit der Funktion `appendData` in die Log-Datei geschrieben. Am Ende des “Logfiles” wird noch mit der Methode `writeTotalTime` die gesamte Zeit für die Simulation in den Log geschrieben. Um die Dateien eindeutig benennen zu können liest der `LogHandler` den Inhalt des Log-Verzeichnisses ein und verwaltet die Anzahl der vorhandenen Log-Dateien. Der Name der Log-Dateien setzt sich nun aus dem Wort “Log” und der Zahl der Log-Dateien zusammen.

NetworkGenerator

Die Erzeugung des Netzwerkes mittels einer Textdatei ermöglicht es ein konkretes Netzwerk zu simulieren, ist für die Simulation von großen Netzwerken ein sehr zeitaufwändiger Ansatz. Aus diesem Grunde wurde zusätzlich zum `InputParser` noch ein `NetworkGenerator` implementiert, der ein zufälliges Netzwerk anhand von vorgegebenen Parametern erzeugt. Für die Generierung wurde das Barabási-Albert Modell [1] verwendet. Bei Knoten welche bereits viele Verbindungen haben werden mit höherer Wahrscheinlichkeit neue Knoten hinzugefügt als bei solchen mit weniger Verbindungen. Falls ein Netzwerk unter 100 Knoten generiert werden soll, so wird das Netzwerk zufällig erzeugt, indem Knoten paarweise per Zufall verbunden werden und jeder Knoten mindestens zwei Verbindungen erhält.

```
/* Generate random node (with 2 connections) */
private boolean generateRandom()
{
    int to2, to, lat;
    float latency, f, t;
    for (int i = 0; i < totalAmount; i++)
    {
        latency = getSecureRandomNumber() * 100;
        lat = (int) latency;
        t = getSecureRandomNumber() * totalAmount;
        f = getSecureRandomNumber() * totalAmount;
        to = (int) t;
        to2 = (int) f;
    }
}
```



```

try
{
    nodesToReach.get(i).addReachable(allNodes.get(to), lat);
    nodesToReach.get(to).addReachable(allNodes.get(i), lat);
    nodesToReach.get(i).addReachable(allNodes.get(to2), lat);
    nodesToReach.get(to2).addReachable(allNodes.get(i), lat);
}
catch (Exception e)
{
    return false;
}
}
return true;
}

```

Die Klasse verwaltet eine `ArrayList` für Knoten und eine weitere für `ReachableList`-Objekte. Diese werden am Ende ins `Network` gespeichert. Auch im `NetworkGenerator` werden die Knoten mittels Reflections erzeugt. Lediglich die Standardknoten werden direkt generiert. Zudem wurde die Gesamtzahl an Verbindungen für einzelne Knoten auf 20 beschränkt. Zu Beginn wird die `ArrayList` für die Knoten mit null-Werten gefüllt. Nun werden die verschiedenen Angreifer-Knoten zufällig über die `ArrayList` verteilt.

```

for (int i = 0; i < amount.size(); i++)
{
    for (int j = 0; j < amount.get(i); j++)
    {
        p = getSecureRandomNumber();
        String type = types.get(i);
        INode node = createObjectNode(type);
        if (node == null)
            throw new NodeTypeNotFoundException();

        float idf = (float) p * totalNodeAmount;
        int id = (int) idf;
        while (allNodes.get(id) != null)
        {
            p = getSecureRandomNumber();
            idf = (float) p * totalNodeAmount;
            id = (int) idf;
        }
        node.setId(id);
        node.setName(types.get(i));
        allNodes.set(id, (Node) node);
        ReachableList rl = new ReachableList(allNodes.get(id));
        nodesToReach.set(id, rl);
    }
}

```

Als Standardknoten im Netzwerk wird der Knotentyp `Node` für das Füllen der verbliebenen null-Werte verwendet.

```

/* Set all null Values in the arraylist to nodes */
for (int i = 0; i < totalNodeAmount; i++)
{
    if (allNodes.get(i) == null)
    {
        Node node = new Node();
        node.setId(i);
        node.setName("Node");
        allNodes.set(i, node);
    }
}

```

```

    ReachableList rl = new ReachableList(allNodes.get(i));
    nodesToReach.set(i,rl);
}
}

```

Wenn die Netzwerkgenerierung abgeschlossen ist, wird noch die Methode `checkBarabasi` ausgeführt, die Knoten mit nur einer Verbindung sucht. Diese bekommen zusätzlich eine zweite Verbindung.

```

/* Check for nodes with only one connection */
private void checkBarabasi()
{
    int to, lat;
    float latency, f, t;
    latency = getSecureRandomNumber() * 100;
    lat = (int) latency;
    t = getSecureRandomNumber() * totalAmount;
    to = (int) t;

    for(int i = 0; i < totalAmount; i++)
    {
        if(nodesToReach.get(i).getLl().size() <= 1)
        {
            nodesToReach.get(i).addReachable(allNodes.get(to), lat);
            nodesToReach.get(to).addReachable(allNodes.get(i), lat);
        }
    }
}

```

Zwar stellt unser Projekt einen Inputparser für die Einpflegung von bestimmten Netzwerken bereit, aber für die Überprüfung in größerem Rahmen, ist der Inputparser nicht mehr geeignet, da die manuelle Spezifikation von 1000 oder mehr Knoten und entsprechenden Kanten sehr Zeitaufwändig sein kann. Hierzu bieten wir eben diese Möglichkeit, ein Netzwerk automatisch generieren zu lassen, wobei vorher für alle Knoten, festgelegt werden kann, wie viele Knoten von diesem Typ generiert werden sollen.

Simulator

Für die Durchführung von einer oder mehreren Simulationen wird die Klasse `Simulator` benötigt. Wichtig ist jedoch, dass eine Instanz des Simulators ausgeführt wird, da es ansonsten zu Datenverlusten und Problemen mit der Simulation kommen kann. Der Simulator verfügt über die zentrale Methode `startSimulation`, die überprüft ob bereits eine Simulation durchgeführt wird und ansonsten eine neue beginnt.

```

public void startSimulation()
{
    running = true;
    try
    {
        while(isRunning())
        {
            running = ev.simulate(network);
        }
    }
    catch (Exception ex)
    {
    }
}

```

```

System.out.println("error at simulator");
ex.printStackTrace();
}
}

```

Die Klasse `Simulator` ist für die Durchführung und den Beginn einer Simulation zuständig. Die Simulation startet durch den Aufruf der Methode. Die Aufgabe der Abarbeitung der Events die sich in der `EventQueue` befinden wird in Folge an den `EventHandler` übergeben. Zudem wird eine `Exception` geworfen, wenn bei der Abarbeitung der Events ein Fehler auftritt.

Algorithm

Das Paket `Algorithm` enthält die Algorithmen für das Routing im Netzwerk. Mit dem Parameter `Algorithm` bei der Netzwerkgenerierung kann der Routing-Algorithmus spezifiziert werden. Dieser wird, wie bereits ausführlich erklärt, mittels Reflections erzeugt. In unserem Simulator ist derzeit der Dijkstra-Algorithmus als einziger Routing-Algorithmus enthalten.

Dijkstra

Die Klasse `Dijkstra` beinhaltet unsere Repräsentation des Dijkstra-Algorithmus. Die Implementierung erfolgte nach der Beschreibung in [9]. Der Algorithmus implementiert das Interface `IAlgorithm` und muss somit die benötigten Methoden überschreiben. Die Speicherung der kürzesten Wege erfolgt über eine `HashMap`, die für jeden Knoten wiederum eine weitere `HashMap` speichert. In dieser ist nun zu jedem Knoten der kürzeste Weg als `LinkedList` von `Nodes` gespeichert. Die Speicherstruktur ist in Abbildung 4.4 dargestellt.

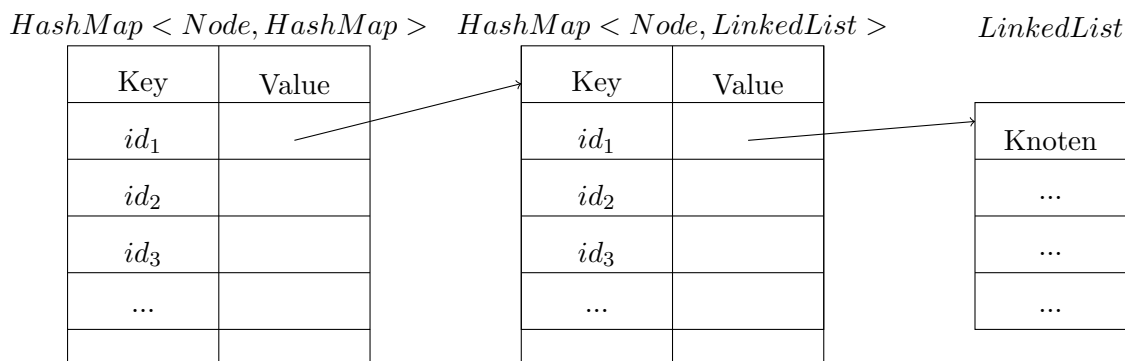


Abbildung 4.4: Speicherstruktur Dijkstra-Algorithmus

Wie in Abbildung 4.4 ersichtlich, verwenden wir eine “HashMap von HashMaps mit LinkedLists” zum Speichern der einzelnen Knoten und ihrer Verbindungen. Wird für einen Knoten der Dijkstra-Algorithmus angewendet, entsteht eine neue `HashMap` und die dazugehörige `LinkedLists` zu allen Knoten.

Protocol

Im Ordner “Protocol” kann ein Anonymisierungsprotokoll für die Simulation implementiert werden. Das Protokoll kann über die Parameter für den Simulator ausgewählt werden. Protokolle

müssen das Interface `IProtocol` implementieren um die einwandfreie Funktionalität des Simulator zu gewährleisten.

Crowds

Diese Klasse beinhaltet die Implementierung der des Crowds Protokolls. Sie enthält zwei wesentliche Methoden für den Ablauf einer Simulation. Zum einen die Methode `executePreSimulation`. In ihr wird das Logging für die Simulation initialisiert und eventuelle Fehler abgefangen. Zum anderen die Methode `executePostSimulation`. Sie ist für die Überprüfung des Theorems 5.2 verantwortlich, die Bedingung hierfür findet sich im Abschnitt 2.1. In dieser Klasse werden alle Knoten ausgegeben die durch kollaborierende Angreifer enttarnt wurden. Zudem wird eine Nachricht ausgegeben, wenn der Startknoten der Kommunikation erfolgreich enttarnt wurde. Weiteres findet eine Überprüfung statt, ob der Startknoten enttarnt wurde, obwohl das Theorem nicht eingehalten wurde. Diese Informationen werden sowohl in der Konsole als auch in der Log-Datei ausgegeben.

Weiters finden sich die drei Methoden `executeTX`, `executeRY` und `executeFinished` in der Klasse. Diese wurden für die Implementierung des Protokolls nicht benötigt, können aber für andere Protokolle genutzt werden, um spezifische Aktionen zu ermöglichen.

Zum Angriffsszenario sind der kollaborierenden Feinden die gemeinsam an der Enttarnung des Initiator arbeiten. Diese Angreiferknoten werden in der Klasse mittels einer `ArrayList` gespeichert. Jeder Angreifer speichert den Knoten von dem er ein Datenpaket erhalten hat:

```
public static ArrayList<Node> collabAL = new ArrayList<>();

/* Add the node that sent the package to the foe, to the list */
public static void addCollabInformation(Node n)
{
    collabAL.add(n);
}

/* probable innocence given, but violated */
if(collabAmount > 0)
{
    if(Network.getInstance().getAllNodes().size() >= (probForward/(probForward-0.5))*(collabAmount +1))
    {
        System.out.println("PROOF: Probable innocence according to the paper violated");
        System.out.println("\nProbable innocence given but violated by collaborating foes\n");
        lh.appendData("Probable innocence according to the paper violated");
        lh.appendData("\nProbable innocence given but violated by collaborating foes\n");
    }
}
```

Data

Im Paket `data` sind die verschiedenen Datenstrukturen die vom Simulator verwendet werden enthalten. Detaillierte Informationen zu den Datenstrukturen sind dem Kapitel 3 zu entnehmen.

Network

Das Netzwerk ist die Hauptklasse für den Simulator und verwaltet alle Knoten sowie die anderen Datenstrukturen. Das Netzwerk wird mittels eines Singleton-Patterns erzeugt und verwaltet um so Probleme bei der Simulation zu verhindern.

Event

Der Simulator verwendet Events um die Kommunikation abzuarbeiten und zu organisieren. Das heißt, das Senden und Empfangen eines Datenpakets und das Ende der Übertragung werden über Events realisiert, welche die Grundklasse Event erweitern. Diese Events werden in einer Liste (`PriorityQueue`) sequentiell hinzugefügt und in der selben Reihenfolge abgearbeitet ("First in, First out"). Auch hier liegt der Vorteil in der Erweiterbarkeit, da für ein neues Event lediglich die Event-Klasse implementiert werden muss.

Abarbeitung

Bei der Simulation wurde bewusst auf Nebenläufigkeit bei der Abarbeitung verzichtet. Es ist allerdings möglich mehrere Simulation nacheinander auszuführen. Dazu muss nur der Wert in der Benutzeroberfläche auf die gewünschte Anzahl der Simulation geändert werden. Zu jeder Simulation wird als Resultat ein Log-File gespeichert in dem man die Ergebnisse der Simulationen analysieren kann.

Events

Die Klasse `Event` ist die Basisklasse. Alle anderen Events erweitern diese Klasse um weitere Funktionen. Events dienen im Simulator der Steuerung der Kommunikation im Netzwerk. Jedes Event verfügt über einen `timestamp`, sowie ein `Layer7Flag` für die Trennung der Kommunikation in die beiden OSI-Schichten die der Simulator unterstützt. Zudem werden in jedem Event der Knoten (`initNode`) der die Übertragung begonnen hat und der Knoten (`receiverNode`) der die Nachricht empfängt gespeichert. Alle weiteren Events die im Simulator Verwendung finden, erben von dieser Klasse.

RXEvent

Ein solches `Event` signalisiert den Empfang eines Datenpaketes. Es speichert die aktuelle Systemzeit und setzt den aktuellen Knoten als neuen Sender und den nächsten Knoten auf dem Weg, als neuen Empfänger.

SimulationFinishedEvent

Dieses Event signalisiert das Ende der Übertragung. Der Simulator wird nun beendet und die Ergebnisse werden in ein Log-File geschrieben.

TXEvent

Dieses `Event` dient dem Versenden eines Datenpaketes. Auf ein `TXEvent` muss immer ein `RXEvent` folgen, dass die `receive` Methode beim Empfänger aufruft.

Data

Wir möchten nun Abschließend auf die wichtigsten Klassen aus dem Paket `data` eingehen, die im wesentlichen die Strukturen eines Netzwerks abbilden. Die Eingangs erwähnten Pakete `view` und `exception` wird hier nichtmehr eingegangen.

Network

Die Klasse **Network** erfüllt verschiedenste Aufgaben. Einerseits werden die existierenden Knoten und alle Datenstrukturen für die Informationsspeicherung verwaltet. Andererseits werden die Parameter für das Netzwerk, das zu verwendende Protokoll und den Routing-Algorithmus gespeichert. Die Ausgabe von wichtigen Informationen in der Konsole wird ebenfalls von dieser Klasse übernommen. Alle Parameter aus dem Inputfile oder der Generierungsoberfläche werden im **Network** in einer **HashMap** gespeichert. In der Klasse **Network** wird der primitive Datentyp **Object** verwendet und mittels Reflections werden die korrekten Objekte erzeugt. Reflections waren für das Projekt notwendig, da vor der Ausführung noch nicht bekannt sein muss welches Protokoll, Algorithmus und Parameter für den Simulator verwendet werden. Als Resultat müssen die Typen der Objekte zur Laufzeit ermittelt werden.

Das verwendete Protokoll und der Algorithmus für das Routing werden als eigene Objekte gespeichert. Die Parameter werden in einer **HashMap** repräsentiert.

```
/* Algorithm and protocol name must correspond to classnames */
public static Object algorithm;
public static Object protocol;

public static HashMap <String, Object> parameter;
```

Zudem verwaltet das Netzwerk noch eine **ArrayList** von **ReachableList**-Objekten. Eine weitere Aufgabe des Netzwerks ist die Ausgabe von relevanten Daten einer Simulation in der Konsole. Hierzu wird die folgende Methode verwendet.

```
/* Output for the console */
public String toString()
{
    String str = "";
    for(int i = 0; i < nodesToReach.size(); i++)
    {
        str += "Node: " + nodesToReach.get(i).getN().getId(); str += "\n";
        for(int j = 0; j < nodesToReach.get(i).getLl().size(); j++)
        {
            ReachableNodes rn= (ReachableNodes) nodesToReach.get(i).getLl().get(j);
            str += " Node: " + rn.getN().getId() + " \tLatency: " + rn.getLatency() + "\n";
        }
        str += "\n";
    }
    return str;
}
```

ReachableNode

Anstelle von Kanten die zwischen den Knoten liegen, wurde in der Implementierung ein sogenanntes **ReachableNode**-Objekt verwendet. Dieses Objekt besteht aus einem **Node** und einem **double**-Wert für die Latenz in Millisekunden. Für die Verwendung im Simulator werden die **ReachableNodes** in einer **ReachableList** verwaltet. Eine detaillierte Erklärung zur Datenstruktur findet sich im Kapitel 3 im Abschnitt 3.

ReachableList

Wir haben bei unserer Implementierung auf Kanten verzichtet, um Verbindungen trotzdem speichern zu können verwenden wir eine **ReachableList**. Die Speicherung der erreichbaren Knoten

erinnert an eine Randknotenliste und wird mittels einer `ArrayList` von `ReachableList` Objekten realisiert. Es ist anzumerken, dass für den erleichterten Zugriff die eindeutige Identifikationsnummer des Knotens mit dem Index in der `ArrayList` übereinstimmt.

In Abbildung 3.1 ist der Zugriff auf die Adjazenzliste grafisch dargestellt.

Paket

Die Klasse `Paket` stellt ein Datenpaket dar, dass Informationen durch das Netzwerk transportiert. Ein Objekt vom Typ `Paket` besteht aus einem `Integer` für die `Id` und einem `String`-Objekt für die `payload`. Das Paket kann bei Angriffsszenarien durch einen Angreifer verändert oder verworfen werden.

5 Beispielimplementierung

THOMAS TRÜGLER

In diesem Kapitel gehen wir auf die Details der Implementierung näher ein und zeigen anhand des Onion-Routing Protokolls, wie ein weiteres Protokoll in den Simulator eingebunden werden kann. Die Details zum Onion-Routing wurden bereits im Grundlagenkapitel 2 besprochen. Wie auch für die Implementierung des Crowds-Protokolls wird bei der Umsetzung des Onion-Routings auf die Verschlüsselung verzichtet, da diese für die Simulationsumgebung nicht relevant ist und zusätzlich Zeit bei der Ausführung benötigt. Bei der Einbettung des Protokolls in den Simulator ist zudem der etwas unterschiedliche Aufbau für Onion-Routing zu beachten. Üblicherweise befinden sich ein Nutzer der die Anfrage an das Netzwerk stellt und der Endserver antwortet außerhalb des Simulations-Netzwerks. Da der Nutzer die mehrfach “verschlüsselte” Nachricht in das Netzwerk schickt und der letzte Knoten des Onion-Netzwerks die entschlüsselte Nachricht überträgt, werden diese Knoten bei der Implementierung außer acht gelassen. Das heißt das jeder Knoten im Onion-Netzwerk als Startknoten und auch als Endknoten fungieren kann. Ein Angreifer der zwischen dem letzten Onion-Knoten und dem Endserver sitzt kann, je nach vorhandener Verschlüsselung, einen erfolgreichen Angriff auf die Daten starten. Da wie schon erwähnt keine Verschlüsselung verwendet wird, werden wir dieses Angriffsszenario im Simulator nicht weiter behandeln.

Wegfindung

Wie bei Crowds werden auch bei diesem Protokoll die Nachrichten über mehrere Knoten geleitet bevor Sie ihr Ziel erreichen. Beim Onion-Routing wird der Weg auf der logischen Ebene zu Beginn der Übertragung festgelegt. Das Routing durch das Netzwerk könnte ebenfalls mittels Dijkstra-Algorithmus, für die kürzesten Wege, erfolgen. Natürlich wäre auch das Verwenden eines anderen Routing-Algorithmus denkbar. Hierzu müsste nur ein neuer Algorithmus im Ordner `algorithmen` implementiert werden. Der Algorithmus kann über den entsprechenden Parameter in der Parameterliste ausgewählt und verwendet werden.

Über die Parameter könnte spezifiziert werden, wie oft ein Paket weitergeleitet werden soll, bevor es an den Empfänger gesendet wird.

Parameter

Auch für dieses Protokoll können die Parameter über die Datei zur Spezifizierung des Netzwerks mitgegeben werden (siehe Abschnitt 4.1) oder im Menüpunkt zur automatischen Generierung des Netzwerks eingestellt werden (siehe Abschnitt 6.2).

Paket

Da auf eine Verschlüsselung verzichtet wird, werden IDs als Header-Namen verwendet um als identifizierende Attribute zu fungieren. Diese Art der Codierung würde zudem die Lesbarkeit erhöhen, bzw. für Menschen erst ermöglichen. Konkret würde die Codierung den Weg des Pakets durch das Netzwerk in Schichten (allerdings in umgekehrter Reihenfolge) schrittweise um das Paket bauen. Also würde für jeden Schritt durch das Netzwerk eine Schicht um das Paket erzeugt werden. Der Aufbau der Codierung ist in der Abbildung 5.1 ersichtlich, wobei id_1 die ID des ersten Knoten auf dem Weg ist und id_n die ID des letzten Knoten auf dem Weg. Ein Vorteil bei dieser Art der Codierung ist auch eine erleichterte Fehlerbehandlung, da mit einer einfachen Abfrage, ob die ID in der Codierung mit der Knoten-ID übereinstimmt, der korrekte Pfad ermittelt werden kann. Der Folgeknoten ist durch diese Codierung ersichtlich und jedem Knoten ist bekannt ob dieser der Empfänger ist oder nicht.

id_1	...	id_n	Paket
--------	-----	--------	-------

Abbildung 5.1: Illustration der schichtweisen Routen-Information in einem Datenpaket

Interfaces

Unter Berücksichtigung der Erweiterbarkeit wurden die Interfaces, für die Protokolle, Algorithmen und Knoten so konstruiert, dass für die weitere Einpflegung von Implementierungen diese als Basis verwendet werden können. Im Fall von Onion-Routing wird das Interface `IProtocol` (siehe Abschnitt 4) als Schnittstelle für das Protokoll verwendet. Das Interface `INode` (siehe Abschnitt 4) wird als Folge für die Knoten (zum Beispiel, `OnionNode`) im Tor-Netzwerk herangezogen.

Knoten

Wie bereits erwähnt bedarf die Implementierung in einer einfachen Form lediglich eines `OnionNode` Knotentyps. Dieser wäre für die Wahl der Route durch das Netzwerk und für den Verbindungsaufbau sowie -abbau zuständig. Im Tor-Netzwerk, dass die bekannteste Umsetzung des Onion-Routings darstellt, wird zudem über diesen Knoten noch der Schlüsselaustausch zwischen dem Startknoten und dem jeweiligen Onion-Knoten durchgeführt. Für eine detaillierte Implementierung kann zudem noch ein `UserNode` eingefügt werden, der außerhalb des Onion-Netzwerks liegt und die Verbindung mit dem Netzwerk herstellt. Da sich hier wenig Angriffsszenarien bieten und die Daten zudem verschlüsselt sein müssten, möchten wir hier nur kurz auf die Möglichkeit hinweisen.

Implementierungsdetails

Hier werden wir auf die beispielhafte Implementierung des Onion-Routing Protokolls näher eingehen. Weiteres werden wir beschreiben wie der bestehende Netzwerksimulator um dieses Protokoll erweitert werden kann, ohne Änderungen an der Grundstruktur der vorhandenen Klassen vorzunehmen. Anhand von Erklärungen und Beispielen werden wir eine mögliche Ausführung erläutern.

Protokoll

Die Kernmethoden eines Protokolls, in unserer Ausführung des Netzwerksimulators, sind *executeTX*, *executeRX*, *executePreSimulation*, *executeFinished* und *executePostSimulation*. Diese Methoden ermöglichen es zu allen relevanten Zeitpunkten im Simulationsablauf protokollspezifische Aktionen auszuführen. Die Methoden *executeTX* und *executeRX* sind nicht mit der tatsächlichen Kommunikationslogik, welche in den Knoten zu finden ist, zu verwechseln. Beim Onion-Routing Protokoll halten die Logik die Knoten im Netzwerk, welche in die Kommunikation eingebunden sind. Eine Möglichkeit wäre es die Wegfindung durch das Onion-Netzwerk in die Methode **executePreSimulation** auszulagern. Somit würde bei der Initialisierung der Simulation (oder genauer des Protokolls) der Weg durch das Netzwerk schon berechnet worden sein.

Knotentypen

Um die Umsetzung neuer Protokolle zu erleichtern, haben wir die gesamte Logik für die Anonymisierung in die Java-Klassen, welche die (Funktionalität der) Knoten repräsentieren ausgelagert. Daher wollen wir nun grob die wichtigsten Knotentypen beschreiben, und wie diese implementiert werden können.

UserNode

Dieser Knoten symbolisiert “normale” Nutzer im Netzwerk. Sie versenden eine mehrfach verschlüsselte Nachricht. Wir würden für eine einfache Implementierung auf einen diesen Knoten verzichten, da die **UserNode** prinzipiell außerhalb des eigentlichen Netzwerks liegt und durch einen **OnionNode** symbolisiert werden kann. Da bei diesem Protokoll die Benutzer im Gegensatz zum Crowds-Protokoll auch nicht aktiv am Routing teilnehmen ist die Rolle der **UserNode** eher gering. Falls man sich zur Implementierung entschließt werden folgende Funktionalitäten benötigt.

- *InitComm*: Methode um die Kommunikation zu starten. Der Knoten baut hierbei eine Verbindung zu einem der **OnionNodes** auf.
- *StartComm*: Methode die das Paket mehrfach verschlüsselt und anschließend weiterleitet. Jeder Knoten entschlüsselt eine Schicht des Pakets und sendet die Nachricht an den nächsten Knoten.
- *Receive*: Funktion um ein Datenpaket zu empfangen und weiterzuverarbeiten.

OnionNode

Diese Art von Knoten bildet den Kern des gesamten Protokolls. Für eine schnellere und einfachere Implementierung kann Sie zudem auch die **UserNode** Klasse beinhalten und die Funktionen übernehmen. Da die gesamte Kommunikation beim Onion-Routing über designierte Knoten läuft, symbolisiert ein Objekt einen Knoten im Netzwerk der die Daten verschlüsselt und an den nächsten Knoten weiterleitet. Ein solcher Knoten bräuchte die folgenden Funktionen für die Kommunikation:

- *FindRoute*: Diese Funktion legt zufällig eine Route durch das Netzwerk fest. Diese Route wird an den Knoten der die Anfrage gesendet hat zurückgeschickt. Dieser kann die Nachricht mit den verschiedenen Schlüsseln verschlüsseln und an den ersten Knoten senden.
- *Transmit*: Wenn ein Datenpaket empfangen wurde muss es auch weitergeleitet werden. Diese Funktion übernimmt das versenden an den nächsten Knoten der im Paket spezifiziert wurde.
- *Receive*: Diese Funktion dient dem Empfangen eines Datenpakets. Wenn das Paket erhalten wurde, wird die Nachricht mit dem Schlüssel des Empfängers entschlüsselt und kann anschließend von der Transmit-Funktion weitergeleitet werden.

Events

Die Events dienen der Steuerung der Kommunikation im Netzwerk und werden benötigt um die Abarbeitung aus einer **PriorityQueue** zu gewährleisten. Diese Events müssen den vorhandenen Entsprechen, da eine Änderung weitgreifende Anpassungen im Simulator mit sich ziehen würde.

InitCommunication und StartCommunication

Die Methode **InitCommunication** wird für die Initialisierung des Netzwerks verwendet. Sie generiert einen zufälligen Weg durch das Netzwerk und liefert diesen zur weiteren Abarbeitung zurück. **StartCommunication** repräsentiert ein Event um die Kommunikation zu Starten. Als Folge dieses Events wird, der erste Sendevorgang an einen Onion-Knoten durchgeführt.

RXEvent

Das Event zum Empfangen eines Datenpakets (**receive**). Auf ein solches Event hin wird im Knoten, der im Event spezifiziert ist, die **receive**-Methode ausgelöst. Handelt es sich beim Knoten nicht um den Empfänger des Paketes wird in weiterer Folge vom Knoten ein Event zum Übertragen des Paketes erzeugt. Auf dieses folgt ein Event zum Empfangen beim Empfänger-knoten.

TXEvent

Das Event zum Übertragen eines Datenpakets (**transmit**). Dieses Event löst im Knoten die Übertragung zum Nachfolgenden aus. Wenn eine Übertragung stattgefunden hat muss ein **receive** im nächsten Knoten folgen.

SimulationFinishedEvent

Dieses Event symbolisiert das Ende der Übertragung. Wird dieses Event aus der `Queue` geholt, so ist die Übertragung beendet und der Simulator schreibt anschließend ein Log-File. Falls mehr als eine Simulation in der grafischen Benutzeroberfläche ausgewählt wurde, so beginnt die nächste Simulation mit einem `StartCommunication`.

6 Simulationsablauf

SIMON TISCHLER

In diesem Kapitel sollen die Funktionen des Simulators anhand von Screenshots der Benutzeroberfläche erklärt werden. Die Benutzeroberflächen finden sich im Simulator im Ordner **view**. Die Benutzeroberflächen wurden mit JavaFX, einer Plattform für die Erstellung von Rich Internet Applications. JavaFX wurde von Oracle spezifiziert und seit *Java 8* fixer Bestandteil des Java Development Kits. JavaFX wird mittels *.fxml-Dateien* beschrieben. Zudem können *.css-Dateien* verwendet werden um eine Benutzeroberfläche optisch zu spezifizieren. Von uns wurde eine fertige Datei gewählt, mit der die Benutzeroberfläche im *Metro-Style* gestaltet wurde.

Start des Simulators

Wenn der Simulator über eine IDE oder die Kommandozeile gestartet wird, öffnet sich zunächst das Startfenster. Die Oberfläche ist in Abbildung 6.1 dargestellt. In diesem Startfenster finden sich zwei Buttons über die die Art der Netzwerkgenerierung gewählt werden kann. Der Nutzer kann sich hier entweder für die Generierung eines Netzwerkes mittels einer Textdatei entscheiden, und das Netzwerk über den Inputparser erzeugen lassen, oder ein Netzwerk vorgegebener Größe automatisch generieren lassen.

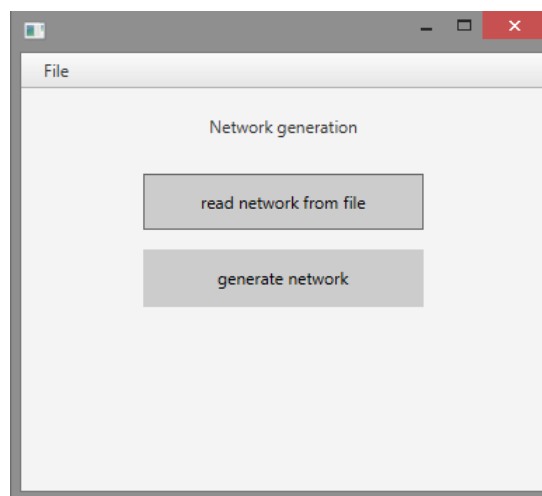


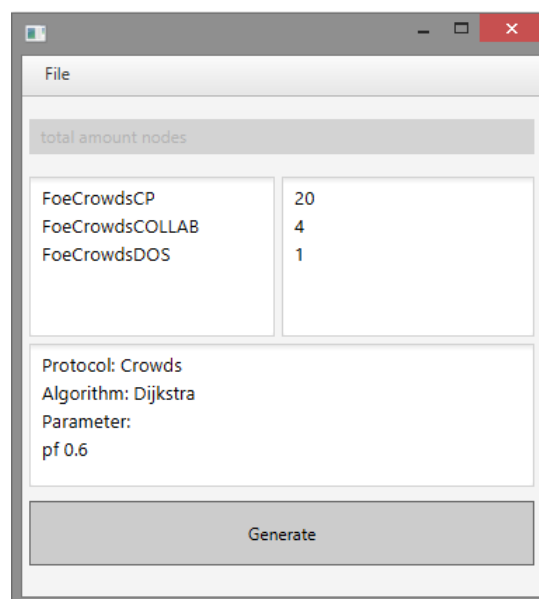
Abbildung 6.1: Startfläche für das Erstellen des Netzwerks.

Einlesen einer Datei

Wenn der/die NutzerIn ein Netzwerk über eine Textdatei erzeugen lässt, so wird ein Dateibrowser geöffnet. In diesem kann er nun zur Datei navigieren und diese Auswählen. Die gewählte Datei wird nun mit dem Inputparser verarbeitet und das Netzwerk erzeugt. Der Aufbau dieser Datei ist im Abschnitt 4 beschrieben. Die Datei muss den dort beschriebenen Anforderungen genügen, damit ein Netzwerk erzeugt werden kann. Wenn die Datei eingelesen wurde, öffnet sich die Oberfläche für das Simulieren eines Netzwerk, die im Abschnitt 6 beschrieben wird.

Netzwerkgenerierung

Wenn sich der/die NutzerIn am Startbildschirm für die zweite Option entscheidet und ein Netzwerk automatisch und zufällig erzeugen lässt so wird kommt er/sie zu einer Benutzeroberfläche für die Generierung eines Netzwerks. Diese Oberfläche ist in Abbildung 6.2 ersichtlich. In der ersten Zeile kann die Gesamtzahl an Knoten für das Netzwerk eingegeben werden. Darunter findet sich ein geteiltes Textfeld für die genaue Spezifikation von anderen Knotentypen als dem originalen *Node*. Für andere Knotentypen muss im linken Textfeld der Knotentyp angegeben werden. Da Reflections verwendet wurden, muss die Klasse im Ordner *Model* gespeichert werden und der Name des Knotentyps muss mit dem Namen der Datei übereinstimmen, wobei auf die Dateiendung *.java* verzichtet wird. Im rechten Textfeld muss nun in der dazugehörigen Zeile die Anzahl an Knoten dieses Typs eingetragen werden. Es ist zu beachten, dass diese Knoten von der Gesamtzahl der Knoten die oben in der Benutzeroberfläche eingetragen wurden abgezogen wird. Im unteren Textfeld müssen nun die für die Ausführung benötigten Parameter eingetragen werden. Nach einem Klick auf den Button *Generate* wird nun das Netzwerk generiert. Unabhängig davon, ob der Nutzer das Netzwerk über eine Knotenliste spezifiziert hat, oder automatisch generieren lässt, findet er sich anschließend in der Oberfläche für die Simulation des Netzwerks wieder.



FoeCrowdsCP	20
FoeCrowdsCOLLAB	4
FoeCrowdsDOS	1

Protocol: Crowds
Algorithm: Dijkstra
Parameter:
pf 0.6

Generate

Abbildung 6.2: Oberfläche für die Erstellung eines Netzwerks.

Simulation

Unabhängig von der Wahl der Netzwerkerzeugung sieht der Nutzer als nächstes die Oberfläche zum Starten der Simulation. In dieser Oberfläche können noch einige Einstellungen vorgenommen werden. In zwei Textfeldern können der Sender und der Empfänger der Konversation eingestellt werden. Weiters kann der Übertragungstext für das Datenpaket festgelegt werden. Für die Überprüfung eines Protokolls ist es oft hilfreich mehrere Simulationen durchführen zu können. Aus diesem Grund gibt es die Option, die Anzahl an Simulationen in einem Textfeld anzugeben. Wenn der Button *start* gedrückt wird, beginnt die Simulation und wichtige Daten werden in der Konsole ausgegeben. Ein Protokoll aller Simulationsschritte findet sich zudem in der erzeugten Protokoll-Datei. Die Erzeugung dieser Datei wird in Abschnitt 4 beschrieben. Der Nutzer kann sich aber auch eine Visualisierung des Netzwerks einblenden lassen indem er auf den Button *show network* klickt. Die Netzwerkvisualisierung wurde mittels des *JUNG-Frameworks* realisiert und wird im Abschnitt 6 beschrieben. Am unteren Ende des Bildschirms gibt es noch zwei Textfelder. Im linken Textfeld werden Daten zu den Verbindungen jedes Knotens angegeben. Auf der rechten Seite wird der Inhalt des Ordners für die Log-Dateien angezeigt.

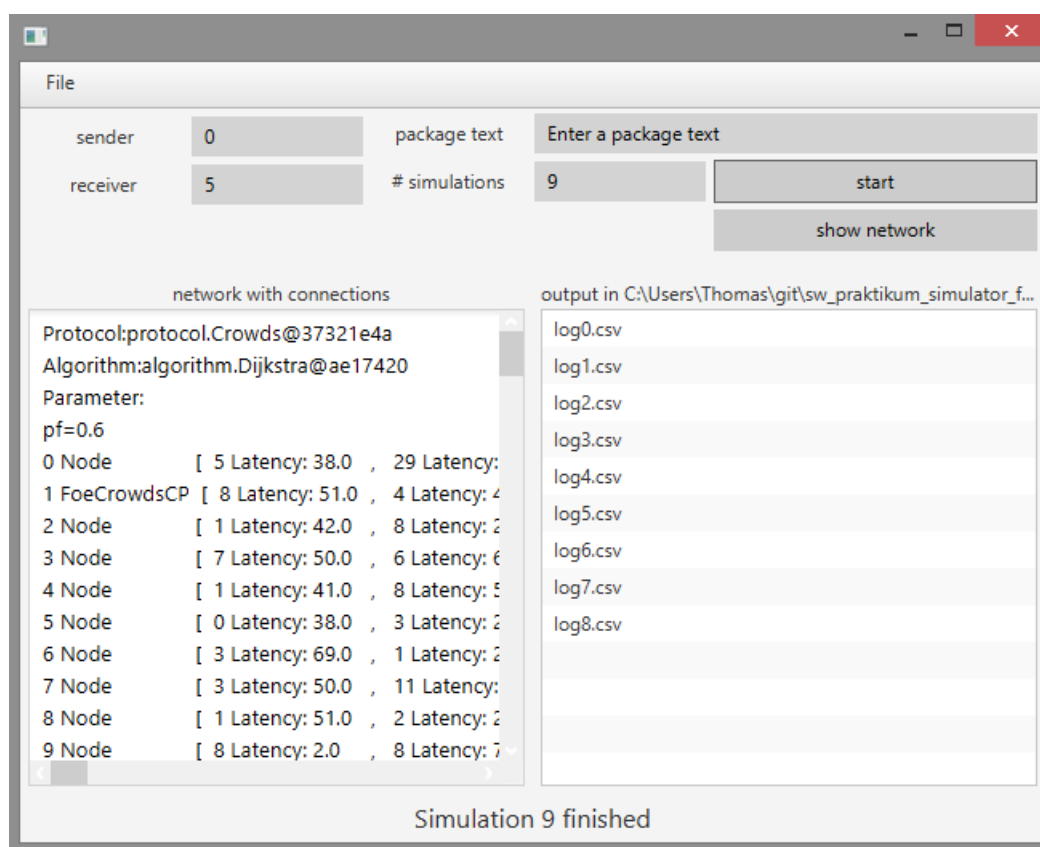


Abbildung 6.3: Simulationsoberfläche für die Kommunikation zwischen Knoten.

Netzwerkvisualisierung

In der Oberfläche für die Simulation findet sich eine Auflistung aller Verbindungen mit Latenzen für jeden Knoten. Mit einer Visualisierung kann sich ein Nutzer einen groben Überblick über

die Knotengrade im Netzwerk verschaffen. Über den Button *show network* gelangt der Nutzer von der Simulationsumgebung aus zur Benutzeroberfläche mit der Netzwerkvisualisierung. Bei der Darstellung des Graphen sind die Knoten kreisförmig nach ihrer eindeutigen Identifikationsnummer angeordnet. Jede Verbindung zwischen zwei Knoten wird als Linie dargestellt. Auf diese Art sind Knoten mit vielen Verbindungen schnell ersichtlich. Wenn der Nutzer auf einen einzelnen Knoten in der Visualisierung klickt, öffnet sich ein kleines Fenster und die Identifikationsnummer, sowie die Anzahl an Verbindungen werden angezeigt. Die Visualisierung des Netzwerks ist zudem in Abbildung 6.4 nochmals illustriert.

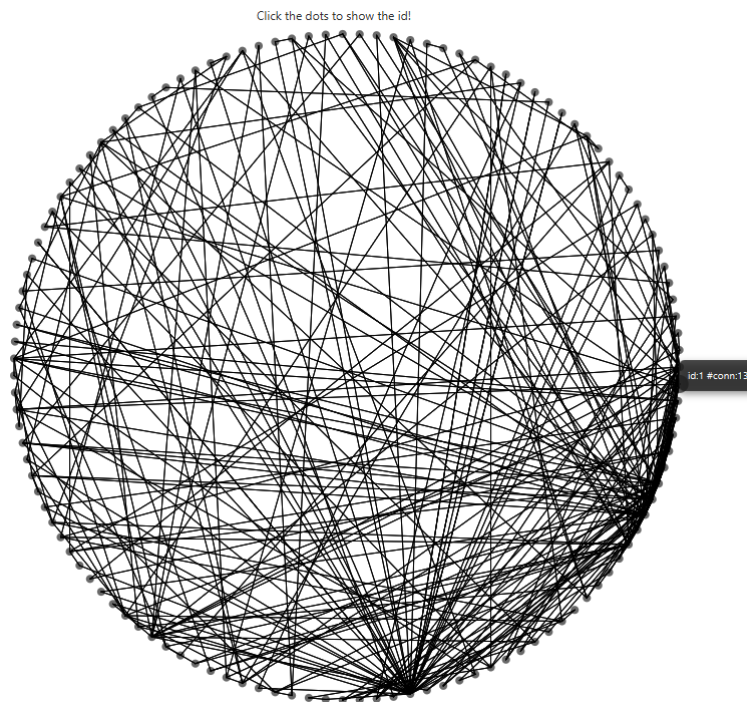


Abbildung 6.4: Veranschaulichung eines (großen) Beispiel-Netzwerks.

7 Erfahrung

THOMAS TRÜGLER, SIMON TISCHLER

In diesem Kapitel soll kurz auf die Erfahrungen, die über den Zeitraum des Softwarepraktikums gemacht wurden, eingegangen werden. Vor allem Probleme, die im Projektverlauf aufgetreten sind werden hier angesprochen, aber auch die Erfolge und der Erfahrungsgewinn.

Erfolge und Erfahrungsgewinn

Durch die ständige Weiterentwicklung wurde die Kommunikation innerhalb des Teams gefördert und die Zusammenarbeit mit anderen Personen trainiert. Durch die Implementierung der zunächst unbekannten Frameworks JavaFX und JUNG, wurden diese im Laufe des Softwarepraktikums durch die ständige Verwendung erlernt. Auch der Umgang mit der Versionsverwaltungssoftware *git* wurde durch dauerhafte Benutzung geschult. Auffällig war, dass oftmals mit der Implementierung begonnen wurde, ohne konkret die Datenstrukturen oder weitere Details vorher abzuklären. Als Folge, sind zum Beispiel, die Datenstrukturen während des Entwicklungsprozesses entstanden. Diese haben sich dann als sehr brauchbar erwiesen. Als der Simulator fertiggestellt war und die Implementierung abgeschlossen, wurde als großer Erfolg die korrekte Überprüfung des Theorems 2 aus dem Crowds-Paper gefeiert.

Änderungen und Erlebnisse im Laufe der Zeit

Da sich im Laufe der Zeit einige Anforderungen änderten, beziehungsweise erst klarer formuliert wurden, gab es einige Änderungen an der Grundstruktur und einzelnen Klassen. Die größten Änderungen wurden aber an der graphischen Benutzeroberfläche vorgenommen.

Änderung von Swing auf JavaFX

Von Beginn an wurde der Simulator in Java mit Swing entwickelt. Allerdings wurde eine erste Benutzeroberfläche nur zu Testzwecken erstellt. Die Portierung des Projektes von Java Swing auf JavaFX erforderte eine Neuerstellung der Benutzeroberflächen. Wir wählten JavaFX aus um das Design des Simulators an einer zentralen Stelle bearbeiten zu können. Diese ist das *Cascading Style Sheet* welches die Designspezifikationen für Buttons, Text Fields, etc. beinhaltet. Die Kommunikation zwischen der Oberfläche und der Logik wird von eigenen Controllern übernommen.

JUNG Framework

Mit der verbesserten Benutzeroberfläche entstand auch der Wunsch, ein Visualisierungstool im Simulator zur Verfügung zu stellen, mit dem ein Netzwerk übersichtlich dargestellt werden sollte. Hier fiel die Wahl auf das *Java Universal Network and Graph Framework*, dass auch detailliert im Abschnitt 6 erläutert wird. Das Framework konnte ohne größere Probleme implementiert werden.

Zeitplan

Die Entwicklung des Simulators begann im März 2014 mit Beginn des Sommersemesters. Am 2. Mai 2014 wurde die Entwicklung auf die Versionsverwaltungssoftware *git* umgestellt.

Während der Implementierung wurde die Benutzeroberfläche des Simulators von Java Swing auf JavaFX umgestellt. Zu diesem Zweck wurde auch ein neues Projekt erstellt. Die Verwaltung mittels *git* erfolgte über einen privaten Server der *gitlab* verwendet.

Da diese Umstellung einen großen Einschnitt darstellt, wurde der Zeitplan in zwei Teile geteilt. Im ersten Teil wird die Entwicklung von Mai bis November 2014 beschrieben, bis zur Umstellung auf die neue Benutzeroberfläche. Im zweiten Teil wird nun die weitere Entwicklung im finalen *git*-Verzeichnis besprochen.

Zeitplan Teil I

In diesem Abschnitt soll auf die Entwicklung des Simulators mit Java Swing als GUI eingegangen werden. Die Entwicklung wurde am 2. Mai begonnen und endete am 30. Oktober mit dem Wechsel auf JavaFX. Eine graphische Übersicht über die Entwicklung findet sich in der Abbildung 7.1. In dem abgebildeten Graphen werden die Commits pro Tag dargestellt.

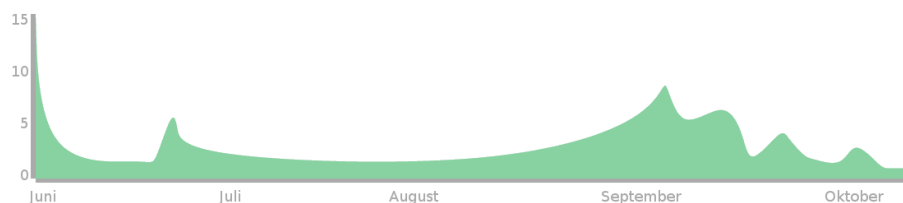


Abbildung 7.1: Arbeitsfortschritt von Mai 2014 bis November 2014. Von *gitlab* automatisch erstellter Graph.

Am Ende des ersten Teils war bereits ein funktionsfähiger Simulator vorhanden, der jedoch wenig mit dem finalen Simulator gemein hat. Der Simulator verfügte noch nicht über Reflections für die Parameterspezifikation. Zudem konnten Netzwerke nur über eine Textdatei erzeugt werden, die Möglichkeit ein Netzwerk automatisch zu generieren war noch nicht gegeben. Die wichtigsten Schritte waren:

- Grundlegende Implementierung
- Datenstrukturen
- Einteilung in das MVC-Pattern

- Inputparser implementiert
- GUI-Prototyp mit Java Swing
- Knotentypen

Zeitplan Teil II

Mit dem Umstieg auf JavaFX und ein neues git-Repository, begann der zweite Teil der Implementierung. Hier wurde vor allem das gesamte Userinterface überarbeitet. Zudem wurde der Simulator überarbeitet um den Anforderungen an das Projekt zu genügen. Eine graphische Veranschaulichung der Entwicklungsergebnisse findet sich in Abbildung 7.2.

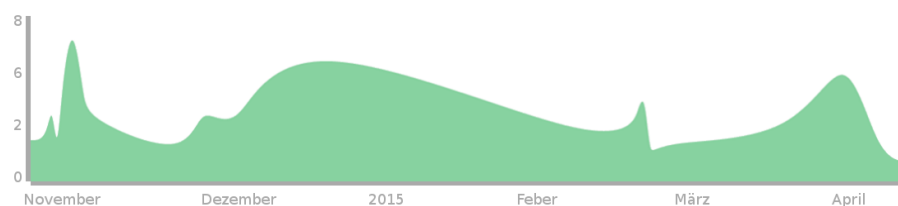


Abbildung 7.2: Arbeitsfortschritt von November 2014 bis April 2015. Von gitlab automatisch erstellter Graph

Die wichtigsten Schritte waren:

- Überarbeitung der GUI
- Netzwerkgenerierung
- GUI mit JavaFX
- Einbau JUNG-Framework für Visualisierung
- Log-Handler
- Reflections

Wichtige Termine

Neben der Implementierung des Simulators finden sich nun die wichtigsten Termine die von den Studenten im Laufe des Softwarepraktikums wahrgenommen wurden. Die Termine sind in Form einer Tabelle angeführt und die Namen der Studenten wurden durch ihre Initialen abgekürzt. In der Tabelle finden sich alle Präsentationstermine sowie die Termine von wichtigen Besprechungen mit unserem Betreuer. Bei diesen Besprechungen wurde der Fortschritt des Simulators und Details zur Implementierung besprochen.

Thomas Trügler: TT

Simon Tischler: ST

Tätigkeit	Datum	Teilnehmer
Vorbesprechung	13. Februar 2014	ST, TT
Offizieller Projektstart.	3. März 2014	ST, TT
Erstpräsentation	4. April 2014	ST, TT
Besprechung Dr. Rass	12. Mai 2014	ST, TT
Projektbeschreibung	30. Mai 2014	ST, TT
Besprechung Dr. Rass	2. Juni 2014	ST, TT
Zwischenpräsentation	3. Juni 2014	ST, TT
Besprechung Dr. Rass	14. November	ST, TT
Abschlusspräsentation	24. März	ST, TT
Abgabe Simulator	24. März	ST, TT
Benotung Softwareprojekt	25. März	ST, TT

Wie in der Tabelle bereits beschrieben, haben wir das Softwarepraktikum am Anfang des Sommersemesters 2014 begonnen. Bereits nach einem Monat fand eine kurze Startpräsentation zum Projekt statt. Es wurden auch regelmäßig Besprechungen zum Projektverlauf mit dem Betreuer abgehalten. In diesen Gesprächen wurden unter anderem technische Details geklärt. Eine Zwischenpräsentation zum aktuellen Projektstand wurde im Juni 2014 abgehalten. Im März 2014 erfolgte die Enpräsentation des Projekts im Rahmen des Privatissimums des Instituts für Systemsicherheit statt. In diesem Rahmen wurde die Implementierung und die zugrundeliegenden Datenstrukturen besprochen und eine kurze Demonstration des Simulators gegeben. Den Schlussspunkt für das Softwarepraktikum setzt die finale Übergabe des Simulators an den Betreuer. Im April und Mai wurde die Dokumentation finalisiert und abgegeben.

Literaturverzeichnis

- [1] Reka Albert and Albert-Laszlo Barabasi. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74:47–97, 2002.
- [2] Miklós Bóna. *A walk through combinatorics : an introduction to enumeration and graph theory*. World Scientific Pub. cop., New Jersey, 2006.
- [3] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–90, feb 1981.
- [4] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.
- [5] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, jun 1962.
- [6] Andreas Pfitzmann and Michael Waidner. Networks without user observability – design options. In Franz Pichler, editor, *Advances in Cryptology – EUROCRYPT’ 85*, volume 219 of *Lecture Notes in Computer Science*, pages 245–253. Springer Berlin Heidelberg, 1986.
- [7] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for web transactions. Technical report, AT&T-Labs, 1997.
- [8] Paul F. Syverson, David M. Goldschlag, and Michael G. Reed. Anonymous connections and onion routing. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, SP ’97, pages 44–, Washington, DC, USA, 1997. IEEE Computer Society.
- [9] Andrew Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 4th edition, 2002.