



**UNIVERSITÄT PADERBORN**

*Die Universität der Informationsgesellschaft*

Fakultät für Elektrotechnik, Informatik und Mathematik

Institut für Informatik

Arbeitsgruppe Theorie verteilter Systeme

## Bachelorarbeit

Gerichtet an die Arbeitsgruppe Theorie verteilter Systeme  
zur Erreichung des Grades

Bachelor of Science

# Schutz und Stabilisierung von Overlay-Netzwerken mithilfe des Relay-Layers

---

Implementierung und Erweiterung des Relay-Modells in Bezug  
auf der Denial of Service Resistenz mit anschließender  
Praxisanalyse des Finite Departure Problems

von

DENNIS SUERMANN  
MATRIKELNUMMER: 7122473

Betreut durch:

Prof. Dr. Christian Scheideler, Prof. Dr. Friedhelm Meyer auf der Heide

Paderborn, 06.10.2021

Der Quellcode und die Dokumentation der Library, welche in dieser Arbeit vorgestellt wird, ist in dem Git-Repository unter <https://git.cs.uni-paderborn.de/denniss2/relaymodel> zu finden.

# Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

Borchen, 06.10.2021

Ort, Datum



Unterschrift



**Zusammenfassung.** Vorherige Arbeiten haben bereits ein theoretisches Modell entwickelt, welches das Finite Departure Problem von verteilten Systemen lösen kann. In dieser Arbeit stellen wir die Implementierung dieses Modells vor und führen anhand dieser Library eine Praxisanalyse des Finite Departure Problems durch. Weiterhin werden Erweiterungen für das Modell entwickelt, welche Distributed Denial of Service Angriffe innerhalb von Overlay-Netzwerken erkennen und verhindern sollen. Dafür wird in der Arbeit ein Algorithmus vorgestellt und in die Library eingebunden. Für die Analyse des Problems werden wir unterschiedliche Knotenmengen mit zufällig falschen Startkonfigurationen simulieren. Die Analyse hat gezeigt, dass die Library mit dem umgesetzten Protokoll Knoten in kurzer Zeit aus einem System ausschließen kann, ohne den Zusammenhang des Graphen zu gefährden. Dabei brauchten Knoten durchschnittlich zwischen fünf und zehn Timeouts bis diese erfolgreich das System verlassen konnten. Weiterhin nahm die Anzahl an Knoten, welche wesentlich länger als zehn Timeouts gebraucht haben, bei steigender Knotenanzahl zu. Auch die Stabilisierung eines Systems benötigte bei steigender Knotenanzahl länger als bei kleineren Knotenmengen.



# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Ziel und Vorgehensweise . . . . .	2
1.2	Struktur der Arbeit . . . . .	2
<b>2</b>	<b>Theoretischer Hintergrund</b>	<b>3</b>
2.1	Basismodell von Overlay-Netzwerken . . . . .	3
2.1.1	Zustände und Netzwerkproblem . . . . .	4
2.2	Selbststabilisierung . . . . .	4
2.2.1	Verbindungsprimitive . . . . .	5
2.2.2	Sortierte Liste . . . . .	6
2.3	Leave Problematik . . . . .	7
2.3.1	Orakel . . . . .	8
2.4	Relay-Modell . . . . .	9
2.4.1	Aufbau des Modells . . . . .	9
2.4.2	Standardaktionen und Nachrichtenverarbeitung . . . . .	12
2.4.3	<i>FDP</i> Lösung durch das Relay-Modell . . . . .	13
2.5	Denial of Service . . . . .	14
<b>3</b>	<b>Implementierung des Relay-Modells</b>	<b>17</b>
3.1	Modul Konfiguration . . . . .	17
3.2	Identifikation von Relays und dem RelayLayer . . . . .	18
3.2.1	RelayLayerId . . . . .	18
3.2.2	RelayId . . . . .	18
3.3	Verbindungsreferenzen . . . . .	19
3.3.1	Ausgehende Referenzen . . . . .	19
3.3.2	Eingehende Referenzen . . . . .	19
3.4	Relay . . . . .	19
3.5	Kommunikation . . . . .	20
3.5.1	Aktionsklassen . . . . .	22
3.6	LinkLayer . . . . .	25
3.6.1	Verbindungen und Pattern . . . . .	25
3.6.2	Ausfallmanagement . . . . .	26
3.6.3	Nachrichtenweiterleitung . . . . .	26
3.6.4	Funktionen außerhalb der Klasse . . . . .	27
3.6.5	Buffer Designentscheidungen . . . . .	28
3.7	RelayLayer . . . . .	29
3.7.1	Grundlegende Änderungen . . . . .	29

3.8	Node . . . . .	31
3.8.1	Benutzung des Knotens . . . . .	31
3.8.2	Änderungen . . . . .	32
3.9	Keys, Logging und Überblick . . . . .	34
<b>4</b>	<b>Denial of Service Erweiterungen</b>	<b>37</b>
4.1	Erkennungsalgorithmen . . . . .	37
4.2	Einbindung in die RelayLayer Klasse . . . . .	40
<b>5</b>	<b>Finite Departure Analyse</b>	<b>43</b>
5.1	Implementierung und Methodik der Analyse . . . . .	43
5.1.1	Zufällige Graphgenerierung . . . . .	46
5.1.2	Methodik . . . . .	48
5.2	Analyseergebnisse . . . . .	49
5.2.1	Stabilisierung des Systems . . . . .	49
5.2.2	Verlassen des Systems . . . . .	50
5.2.3	Einordnung der Ergebnisse . . . . .	52
5.2.4	Erwähnenswerte Beobachtungen . . . . .	53
<b>6</b>	<b>Ausblick und Zusammenfassung</b>	<b>55</b>
6.1	Ausblick . . . . .	55
6.2	Zusammenfassung . . . . .	56
	<b>Literaturverzeichnis</b>	<b>57</b>



# Einleitung

In verteilten Systemen ist das Verlassen eines Systems ein großes Problem für die Teilnehmer. Je größer das System, desto höher ist die Wahrscheinlichkeit, dass sich die Teilnehmer des Systems verändern [SS18]. Somit wollen neue Teilnehmer dazu kommen und bereits involvierte das System verlassen. Im klassischen Model von verteilten Systemen kann das Problem nur mithilfe von Orakeln gelöst werden, welche den gesamten Systemzustand kennen. Zum Lösen des Problems ohne ein globales Orakel entwickelten Setzer und Scheideler [SS18, Set20] ein neues Modell für Overlay-Netzwerke, welches das Verlassen eines Systems ermöglichen soll, ohne dass die Funktionalität des Netzwerks beeinträchtigt wird. Das theoretische Konzept behandelt Verbindungen über sogenannte Relays, welche wiederum für jeden Teilnehmer im System durch einen entsprechenden Relay-Layer verwaltet und kontrolliert werden. Diese Relays beinhalten grundlegende Informationen über die Konnektivität des Teilnehmers und dienen als Schnittstelle zwischen zwei Teilnehmern. Sie enthalten Informationen über eingehende und ausgehende Verbindungen zu anderen Relays. Durch die kontrollierte Verbindung über Relays können Informationen, welche normalerweise ein Orakel benötigen, lokal abgerufen werden. Dadurch ist es möglich Protokolle zu entwickeln, welche das Verlassen des Systems mit den Informationen der Relays gewährleisten können.

Durch die steigende Anzahl an Internetgeräten, welche schätzungsweise bis 2023 mehr als die dreifache Anzahl der Weltbevölkerung annimmt, steigen auch die Angriffe auf einzelne Teilnehmer [Cis20]. Vor allem die Distributed Denial of Service (DDoS) Angriffe steigen laut aktuellen Schätzungen voraussichtlich von 2018 mit 7,9 Millionen Angriffen bis 2023 auf 15,4 Millionen an [Cis20]. Deshalb gilt es eine Methode zu finden, welche diese Angriffe erkennen und verhindern kann. Das Problem in verteilten Systemen ist, dass Teilnehmer nicht entscheiden können, welche Knoten mit diesen kommunizieren können und welche nicht. Dadurch können Referenzen von einem bestimmten Teilnehmer an andere weitergeleitet werden, sodass wiederum DDoS Angriffe gegen den bestimmten Teilnehmer durchgeführt werden können. Auch dafür wurden in dem neuen Modell von Scheideler und Setzer [SS18, Set20] Methoden hinzugefügt. Durch diese Restriktion ist es möglich, bereits aufgebaute eingehende Verbindungen zu verbieten und zu entfernen. Dadurch können Algorithmen entwickelt werden, welche bei einem potenziellen DDoS Angriff bestimmte Verbindungen verbieten oder entfernen. Bevor die Verbindungen jedoch eingeschränkt werden können, bleibt es zu überlegen, wie Angriffe innerhalb des Modells erkannt werden können. Generell wurde das theoretische Konzept bereits studiert, jedoch wurde das Modell noch nicht implementiert und in der Praxis analysiert. Das Implementieren, Analysieren und Erweitern durch DDoS Erkennungsalgorithmen bildet die Basis dieser Arbeit.

## 1.1 Ziel und Vorgehensweise

Die Arbeit teilt sich in drei Teile auf. Im ersten Teil soll das theoretische Modell des Relay-Layers als nutzbare Library implementiert werden. Nachfolgend kann anhand dieser Library eine Analyse durchgeführt werden. Die Analyse soll zeigen, wie gut das System damit zurechtkommt, wenn Teilnehmer ein System verlassen möchten. Für den letzten Teil sollen Erweiterungen für das Modell entwickelt werden, welche DDoS Angriffe effizient erkennen und verhindern können.

**Library:** Für die Implementierung wird eine Library in Python erstellt und später vorgestellt. Diese Library soll dann dafür verwendet werden, vorhandene Protokolle mit dem Relay-Modell nutzbar zu machen. Weiterhin sollen Methoden erstellt werden, um Knotenkonfigurationen zu erstellen und analysieren zu können.

**DoS Erweiterungen:** Für die Erweiterungen werden wir später die Übertragungen der einzelnen Verbindungen überwachen, um so die potenziellen Angriffe zu identifizieren und notwendige Maßnahmen bei Erkennung durchzuführen. Dafür sollen Algorithmen entworfen werden, welche erstens die Verbindungen überwachen und zweitens Angriffe erkennen können.

**Analyse:** Für die Analyse wird das BuildList Protokoll [FSS20] einer Sortierten Liste in verteilten Systemen so angepasst, dass dieses für Verbindungen zu anderen Knoten das implementierte Relay-Modell verwendet. Mit dieser angepassten Implementierung werden anschließend unterschiedlich zufällige Startkonfigurationen simuliert und anhand der Ergebnisse analysiert. Die Startkonfigurationen können beliebig falsch sein und müssen nicht eine bereits vollständig geformte Sortierte Liste bilden. Die einzige Voraussetzung der Konfiguration wird sein, dass diese schwach zusammenhängend ist und somit alle Knoten innerhalb des Systems erreicht werden können. Zu der Startkonfiguration wird es Knoten innerhalb der Simulation geben, welche das System verlassen wollen und dieses anhand des Protokolls durchführen.

## 1.2 Struktur der Arbeit

Im nachfolgenden Abschnitt 2 werden zuerst alle theoretischen Grundlagen für die Arbeit erklärt und wiedergegeben. Dazu zählen grundlegende Definitionen und Konzepte von verteilten Systemen. Auch wird das Relay-Modell vorgestellt, auf dem die Implementierung basiert. Danach wird in Abschnitt 3 die implementierte Library näher erläutert. Dabei wird der grundlegende Aufbau dieses Frameworks erklärt und die wichtigsten Funktionen näher beschrieben. Zudem wird erläutert, wie selbststabilisierende Protokolle mit der Library implementiert werden können, damit diese den Relay-Layer und das zugehörige Protokoll benutzen. Nachdem die Library vorgestellt wurde, werden im Abschnitt 4 Algorithmen vorgestellt, welche DoS Angriffe erkennen und verhindern sollen. Zudem wird erläutert, wie diese Algorithmen innerhalb des Relay-Layers eingebunden und in der Library genutzt werden können. Im Abschnitt 5 wird die durchgeführte Analyse näher beschrieben. Dabei wird zuerst die Implementierung der Analyse vorgestellt und die Methodik näher erläutert. Anschließend werden die Ergebnisse vorgestellt und kritisch bewertet. Zuletzt wird die Arbeit in Abschnitt 6 zusammengefasst. Dabei werden offene Fragestellungen vorgestellt und die wesentlichen Ergebnisse der Arbeit kurz wiedergegeben.

## Theoretischer Hintergrund

Nachfolgend werden grundlegende Begrifflichkeiten und theoretische Konzepte, auf die diese Arbeit aufbaut, näher erläutert. Dafür werden zuerst Grundlagen von Overlay-Netzwerken definiert, mit denen danach Selbststabilisierung definiert werden kann. Weiterhin werden die Problematiken beim Verlassen von Systemen erläutert, welche in dieser Arbeit näher betrachtet und simuliert werden. Nachdem alle grundlegenden Definitionen geklärt wurden, wird das Modell, welches als Grundlage der Implementierung dieser Arbeit dient, vorgestellt. Am Ende dieses Kapitels werden Denial of Service (DoS) Attacken, welche wir in dieser Arbeit durch Erweiterungen im Relay-Modell verhindern wollen, näher beschrieben.

### 2.1 Basismodell von Overlay-Netzwerken

Die nachfolgenden Definitionen stammen aus [FSS20] und werden zusammengefasst wiedergegeben. Ein Overlay-Netzwerk wird mithilfe von einem gerichteten Graphen  $G = (V, E)$  dargestellt, wobei  $n = |V|$  die Anzahl an Knoten innerhalb des Netzwerkes ist. Jeder Teilnehmer dieses Netzwerkes wird durch einen Knoten  $v \in V$  dargestellt, welcher einen eindeutigen nicht veränderbaren Identifier  $v.id \in \mathbb{N}$ , lokale Variablen und einen Channel  $v.Ch$  besitzt. Die Kanten  $(u, v)$  des Graphen repräsentieren Referenzen eines Knoten  $v$  in lokalen Variablen innerhalb des Knoten  $u$ . Durch diese Referenzen können Nachrichten an andere Knoten gesendet werden, indem der Knoten  $u$ , welcher eine Nachricht an  $v$  senden möchte, die Nachricht in den Channel  $v.Ch$  von  $v$  einfügt. Dieser Channel beinhaltet eingehende Nachrichten des bestimmten Knoten. Bei diesen Nachrichten, welche an andere Knoten geschickt werden, handelt es sich standardmäßig um Aktionsaufrufe des anderen Knoten.

Aktionen sind entweder in der Form  $\langle label \rangle(\langle parameters \rangle) : \langle command \rangle$ , wobei  $label$  den Name der Aktion, die  $parameters$  die Informationen definieren, welche für die Aktion benötigt werden, und  $command$  die Programmstatements, welche durch die Aktion aufgerufen werden, definiert. Die zweite Art von Aktionen sind in der Form  $\langle label \rangle : (\langle guard \rangle) \rightarrow \langle command \rangle$ .  $label$  und  $command$  definieren hier dieselben Eigenschaften wie bei dem Aktionstyp zuvor, wobei der Guard eine Bedingung definiert, unter der die Aktion ausgeführt wird. Ein besonderes Beispiel dafür ist die *timeout* Aktion, welche als Guard *true* gesetzt hat, wodurch die Aktion periodisch ausgeführt wird. Wie wir später sehen werden, ist diese besondere Funktion für die Overlay-Netzwerke wichtig, da dadurch selbststabilisierende Protokolle entworfen werden können. Jede Nachricht, welche sich innerhalb des Channels eines Knotens beziehungsweise eines Prozesses befindet, kann irgendwann ausgeführt werden. Auch die Reihenfolge der Ausführungen

von Aktionen ist nicht geregelt, sodass es nur wichtig ist, dass eine Aktion ausgeführt wird. Dies erlaubt Verzögerungen der Ausführungen von Aktionen oder Verzögerungen beim Weiterleiten der Nachrichten.

Es wird erlaubt, dass Nachrichten nicht nach dem FIFO Prinzip verarbeitet werden. Da es in der Praxis einfacher ist, die Nachrichten in einen Buffer einzufügen und dann nach der Reihenfolge abzuarbeiten, werden wir später trotzdem einen FIFO Ansatz in der Implementierung zum Verarbeiten von Nachrichten verwenden (3.6).

Jeder Knoten, der die oben beschriebenen Eigenschaften besitzt und Aktionen ausführen kann, wird auch als Prozess bezeichnet.

### 2.1.1 Zustände und Netzwerkproblem

Jeder Prozess eines Overlay-Netzwerkes besitzt lokale Variablen und Aktionen. Der Prozesszustand wird durch die Inhalte der lokalen Variablen und anderen Informationen, die sich aktuell im Prozess befinden, beschrieben. Ausgenommen davon sind Nachrichten, welche noch nicht an dem Prozess angekommen sind oder bereits an andere Prozesse gesendet wurden.

Der Netzwerkzustand des Overlay-Netzwerks wird durch alle Nachrichten und anderweitigen Informationen beschrieben, welche zu einem bestimmten Zeitpunkt noch nicht bei dem entsprechenden Prozess angekommen sind und somit noch zugestellt werden müssen.

Der Systemzustand wird dann durch alle Prozesszustände der im Netzwerk befindlichen Prozesse und dem Netzwerkzustand des Netzwerks definiert.

Es ist möglich, dass initiale Systemzustände korruptierte Informationen innerhalb der Zustände besitzen. Sollten korruptierte Informationen oder falsche Kanten, welche fehlerhaft im Bezug auf der zugrundeliegenden Topologie sind, innerhalb der Zustände existieren, ist der Systemzustand illegal. Ein legaler Systemzustand besitzt hingegen keine korruptierten Informationen in beliebigen Prozesszuständen oder im Netzwerkzustand. Weiterhin müssen, damit der Zustand legal ist, die Kanten und Knoten des Graphen die gewünschte Topologie bilden. Ein Netzwerkproblem  $P$  hat als Grundlage einen initialen Systemzustand  $S_0$  gegeben. Gesucht wird ein legaler Systemzustand  $S(S_0)$ , welcher durch Ausführungen von Aktionen erreicht werden kann [Sch20].

## 2.2 Selbststabilisierung

Bevor die Selbststabilisierung formuliert wird, benötigen wir zuerst die Definition des schwachen Zusammenhanges eines Graphen. Diese Eigenschaft ist grundlegend für die Konnektivität eines Overlay-Netzwerks und wird später benötigt. Die nachfolgenden Definitionen basieren auf den Definitionen von West 2001 [Wes01].

**Definition 2.1** (Zusammenhängender ungerichteter Graph [Wes01]). *Ein ungerichteter Graph  $G = (V, E)$  heißt zusammenhängend, wenn für jedes Knotenpaar  $u$  und  $v$  ein Pfad in  $G$  existiert.*

**Definition 2.2** (Ungerichteter Graph eines gerichteten Graphen [Wes01]). *Der ungerichtete Graph  $\mathcal{G}(D)$  eines gerichteten Graphen  $D$  wird erzeugt, indem jede gerichtete Kante durch eine ungerichtete Kante ausgetauscht wird.*

**Definition 2.3** (Zusammenhängender gerichteter Graph [Wes01]). *Ein gerichteter Graph  $D = (V, E)$  ist genau dann schwach zusammenhängend, wenn der zugrundeliegende ungerichtete Graph  $\mathcal{G}(D)$  von  $D$  zusammenhängend ist. Ein gerichteter Graph  $G = (V, E)$  ist stark zusammenhängend, wenn für jedes Knotenpaar  $u$  und  $v$  ein Pfad in  $G$  von  $u$  nach  $v$  existiert.*

**Definition 2.4** (Zusammenhangskomponente [Wes01]). *Eine Zusammenhangskomponente eines ungerichteten Graphen  $G = (V, E)$  ist ein maximal zusammenhängender Teilgraph von  $G$ . Eine schwache Zusammenhangskomponente eines gerichteten Graphen  $G = (V, E)$  ist ein maximaler Teilgraph von  $G$ , der schwach zusammenhängend ist.*

Nun können wir Selbststabilisierung definieren. Dies beschreibt das grundlegende Konzept, durch das illegale Zustände von Systemen nach endlicher Zeit in legale Zustände überführt werden. Zugrundeliegend ist immer ein Netzwerkproblem  $P$  gegeben.

Dijkstra definierte 1974 das grundlegende Konzept der Selbststabilisierung [Dij74], welches nachfolgend 2020 von Feldmann et al. in „Survey on Algorithms for Self-stabilizing Overlay Networks“ weiter festgelegt wurde [FSS20]. Die nachfolgende Definition stammt aus der Studie von Feldmann et al. und der Vorlesung „Verteilte Datenstrukturen und Algorithmen“ aus dem Sommersemester 2020 [Sch20].

**Definition 2.5** (Selbststabilisierung [Dij74, Sch20, FSS20]). *Ein Protokoll heißt selbststabilisierend bezüglich  $P$ , wenn die folgenden Anforderungen erfüllt sind:*

1. **Konvergenz:** *Für alle initialen Systemzustände gewährleistet das Protokoll, dass das System in endlicher Zeit einen legalen Systemzustand erreicht.*
2. **Abgeschlossenheit:** *Für alle legalen Systemzustände ist auch jeder Folgezustand legal.*

Bei verteilten Systemen, welche wir in dieser Arbeit betrachten, ist es wichtig, dass die Knoten für die Umsetzung eines selbststabilisierenden Protokolls miteinander kommunizieren müssen. Um das zu gewährleisten, muss der zugrundeliegende Graph mindestens schwach zusammenhängend sein. Denn sollten mehrere schwache Zusammenhangskomponenten in dem System existieren, könnten diese nicht miteinander kommunizieren, da keine Verbindung zwischen diesen besteht. Infolgedessen könnte das System nicht in einen legalen Zustand überführt werden, da sich ansonsten zwei legale Untersysteme bilden würden [FSS20]. Um dies sicherzustellen, werden nachfolgend Operationen vorgestellt, die den schwachen Zusammenhang des Systems bewahren.

### 2.2.1 Verbindungsprimitive

Die Verbindungsprimitive sind Operationen, welche den Zusammenhang eines Systems beibehalten sollen [FSS20]. Wie in Abbildung 2.1 zu sehen, gibt es die Primitive Vorstellung (Introduce), Weiterleitung (Forward), Verschmelzung (Merge) und Umkehrung (Invert). Die durchgezogenen Linien in der Abbildung sind explizite Kanten und die gestrichelten Linien sind implizite Kanten. Implizite Kanten sind Verbindungen von Knoten, welche noch nicht explizit in Variablen gespeichert sind, sondern sich noch auf dem Weg zu dem Knoten befinden [FSS20]. Bei der Vorstellung gibt der eigene Knoten  $u$  einem anderen Knoten  $v$  eine bestimmte Referenz zu  $w$  weiter [FSS20]. Die Weiterleitung besitzt das gleiche Vorgehen, jedoch wird im Knoten  $u$ , welcher die Referenz weiterleitet, die Referenz zu  $w$  entfernt [FSS20]. Bei der Verschmelzung, welche auch Merge genannt wird, werden zwei Referenzen vom gleichen Knoten zusammengefügt [FSS20]. Zuletzt erreicht die Umkehrung, dass eine Verbindung von  $u$  nach  $v$  umgekehrt wird, sodass  $u$  keine Referenz mehr zu  $v$  besitzt [FSS20]. Stattdessen existiert jedoch eine Referenz von  $v$  nach  $u$ . Die Umkehrungsoperation ist zudem die einzige Operation von den vier genannten, welche einen Knoten unerreichbar machen kann [FSS20].

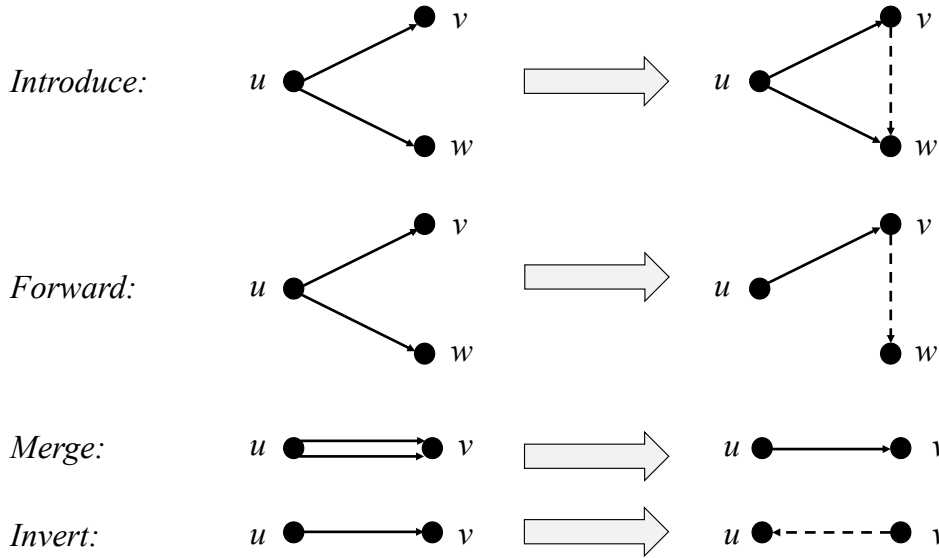


Abbildung 2.1: Verbindungsprimitive (stammt aus [FSS20])

Weiterführend können wir sagen, dass die Primitive Vorstellung, Weiterleitung und Verschmelzung sogar jeden schwach zusammenhängenden Graphen in einen stark zusammenhängenden Graphen formatieren können [FSS20]. Dieses Verhalten wird schwach universell genannt und nachfolgend in einem Satz festgehalten.

**Satz 2.6** (Schwach universell [FSS20]). *Die drei Primitive Vorstellung, Weiterleitung und Verschmelzung sind schwach universell. Das heißt, dass man mit ihnen von jedem schwach zusammenhängenden Graphen  $G = (V, E)$  zu jedem stark zusammenhängenden Graphen  $G' = (V, E')$  gelangen kann.*

Alle vier Primitive zusammen können hingegen jeden schwach zusammenhängenden Graphen in jeden anderen schwach zusammenhängenden Graphen transformieren und sind somit universell [FSS20]. Diese Eigenschaft legen wir nachfolgen in dem Satz fest.

**Satz 2.7** (Universell [FSS20]). *Die vier Primitive Vorstellung, Weiterleitung, Verschmelzung und Umkehrung sind universell. Das heißt, dass man mit ihnen von jedem schwach zusammenhängenden Graphen  $G = (V, E)$  zu jedem schwach zusammenhängenden Graphen  $G' = (V, E')$  gelangen kann.*

Beide Sätze wurden unter anderem in [FSS20] festgelegt und in [Sch20] übersetzt wiedergegeben.

Mit den Verbindungsprimitiven können wir nun selbststabilisierende Protokolle definieren, welche eine bestimmte Topologie formen.

### 2.2.2 Sortierte Liste

Die Sortierte Liste ist eine Topologie, in der jeder Knoten  $u$  mit der ID  $u.id$  eine Referenz zu dem Knoten  $v$  mit  $u.id < v.id$  und eine Referenz zu dem Knoten  $w$  mit  $u.id > w.id$  besitzt, falls diese Knoten existieren [FSS20]. Dabei ist der Knoten mit der nächstkleineren ID der linke Nachbar und der mit der nächstgrößeren ID der rechte Nachbar [FSS20]. Somit besitzt jeder Knoten,

abgesehen vom ersten und letzten Knoten der Liste, einen linken und einen rechten Nachbarn, welche in den lokalen Variablen  $u.left$  und  $u.right$  gespeichert werden [FSS20]. Da jeder Knoten nur eine lokale Ansicht des Systems besitzt, muss ein Protokoll entwickelt werden, das eine Sortierte Liste bilden kann. Dafür formulierten Feldmann et al. [FSS20] das BuildList Protokoll, welches genau dies erreicht. Ein möglicher Aufbau der Liste, welcher nach dem Ausführen des BuildList Protokolls entsteht, kann in der Abbildung 2.2 betrachtet werden. Dabei besitzt der Knoten 3 zum Beispiel eine Referenz zu dem linken Nachbarn mit der ID 1, da dieser der Knoten mit der nächstkleineren ID ist, und eine Referenz zu dem rechten Nachbarn mit der ID 4.

Da die genaue Funktionsweise des Protokolls keinen Mehrwert für diese Arbeit bietet, wird das Vorgehen nicht explizit erklärt und auf die Referenzen [FSS20] und [Sch20] verwiesen.

Wie später in Abschnitt 5.1 beschrieben wird, nutzen wir die Sortierte Liste als grundlegende Topologie, um die Analyse des Relay-Modells durchzuführen, welches nachfolgend in 2.4 erklärt wird.

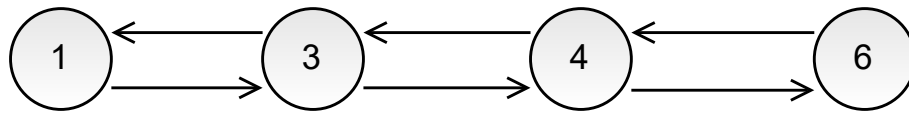


Abbildung 2.2: Möglicher Aufbau einer Sortierten Liste nach dem BuldList Protokoll

## 2.3 Leave Problematik

Zu den in Abschnitt 2.1 beschriebenen Aktionen kommen zusätzlich zwei weitere Standardaktionen, welche das Verlassen eines Systems ermöglichen können. Die erste Aktion ist der *exit* Befehl, welcher einen Knoten in den *exit state* versetzt. Dadurch ist der Knoten „tot“ und kann keine Aktionen mehr ausführen [FKN<sup>+</sup>14, Sch20, SS18]. Mit dem *sleep* Befehl wird ein Knoten in den *sleep state* gebracht, was bedeutet, dass der Prozess selbst keine Aktionen, welche den Guard auf *true* gesetzt haben, ausführt. Sollte jedoch eine Nachricht im Channel des Prozesses auftauchen, erwacht der Knoten und verlässt den *sleep state* [FKN<sup>+</sup>14, Sch20, SS18]. Nach diesen Eigenschaften kann ein toter Knoten nicht wieder aufwachen, wohingegen ein schlafender Knoten durch eine Nachricht wieder aufgeweckt werden kann [FKN<sup>+</sup>14]. Zudem gilt ein Knoten  $u$  als „scheintot“, wenn  $u$  schläft, der Channel von  $u$  leer ist und jeder Knoten, welcher einen gerichteten Pfad zu  $u$  besitzt, auch schläft und keine Nachricht im entsprechenden Channel besitzt [FKN<sup>+</sup>14, Sch20].

Sollte ein Knoten das System verlassen wollen, wird die Operation *Leave* aufgerufen, welche den *leaving* Status des Knotens auf *true* setzt. Das selbststabilisierende Protokoll soll dann das Verlassen des Systems ermöglichen [Sch20].

Aufgrund der Möglichkeit das System verlassen zu können muss wiederum der Begriff des legalen Systemzustands erweitert werden. Das System ist in einem legalen Status, wenn (1.) jeder im System bleibende Knoten wach ist, (2.) jeder Knoten, der das System verlassen möchte, entweder schlafen gelegt wurde oder das System mit *exit* verlassen hat und (3.) für jede schwache Zusammenhangskomponente des initialen Systems bleibende Knoten innerhalb der Komponente weiterhin eine schwache Zusammengangs-komponente bilden [FKN<sup>+</sup>14]. Mit diesen Voraussetzungen wurden dann von Foreback et al. das *Finite Departure Problem* (*FDP*) und das *Finite Sleep Problem* (*FSP*) vorgestellt.

Das *FSP* definiert, dass das System in endlicher Zeit einen legalen Zustand erreichen soll, wenn es nur den *sleep* Befehl zur Verfügung hat [FKN<sup>+</sup>14, Sch20]. Im Gegensatz zum *FSP* ist für das *FDP* nur der *exit* Befehl verfügbar, mit dem das System in endlicher Zeit in einem legalen

Zustand landen soll [FKN<sup>+</sup>14, Sch20]. Das  $\mathcal{FSP}$  kann durch ein selbststabilisierendes Protokoll, wie Foreback et al. in ihrer Arbeit zeigen, gelöst werden. Jedoch ist es für das  $\mathcal{FDP}$  nicht möglich, ohne den Gebrauch von einem Orakel ein selbststabilisierendes Protokoll zu entwickeln [FKN<sup>+</sup>14].

### 2.3.1 Orakel

Ein Orakel  $\mathcal{O}$  ist ein Prädikat, welches von dem Systemzustand und von einem Knoten  $v$ , der das Orakel abfragt, abhängt [FKN<sup>+</sup>14, Sch20]. Genauer gesagt sind die Orakel in der Form  $\mathcal{O} : \mathcal{PG} \times \mathcal{P} \rightarrow \{true, false\}$ , wobei  $\mathcal{PG}$  die Menge von Prozessgraphen und  $\mathcal{P}$  die Menge von Prozessen ist [FKN<sup>+</sup>14]. Die Ausgabe eines Orakels ist entweder *true* oder *false* und gibt somit Auskunft, ob ein bestimmter Status im System für einen Graphen und Prozess erfüllt ist oder nicht. Bevor wir jedoch die Orakel, welche gebraucht werden, um das  $\mathcal{FDP}$  in einem selbststabilisierenden Protokoll lösen zu können, definieren können, benötigen wir noch folgende Begriffe: Eine Kante  $(v, w)$  mit  $v \neq w$  ist *relevant* für einen Prozess  $u$ , wenn  $u = w$  und  $v$  nicht „tot“ ist, oder wenn die Kante in einer Nachricht existiert, welche die ID von  $w$  beinhaltet und für  $u$  bestimmt ist [FKN<sup>+</sup>14]. Sollte diese Voraussetzung nicht erfüllt sein, ist die Kante *irrelevant* für  $u$  [FKN<sup>+</sup>14]. Ein Orakel ist *id-sensitive* für einen Prozess  $u$ , wenn die Ausgabe des Orakels von *relevanten* Kanten von  $u$  abhängt [FSS20]. Ein Orakel heißt *strictlyid-sensitive*, wenn die Ausgabe des Orakels für alle Prozesse  $u$  nur von den *relevanten* Kanten von  $u$  abhängt, sodass *irrelevante* Kanten ignoriert werden [FKN<sup>+</sup>14].

Für das  $\mathcal{FDP}$  brauchen wir ein Orakel, welches Auskünfte über das mögliche Verlassen eines Knoten geben kann. Somit schränken wir das Verlassen eines Knotens dahingehend ein, dass der Knoten den *exit* Befehl nur ausführen darf, wenn das zugrundeliegende Orakel wahr ist [FKN<sup>+</sup>14].

Für dieses Vorhaben definieren Foreback et al. mehrere Orakel:

$\mathcal{EXIT}$ : Ist wahr, wenn ein Prozess  $u$  das System sicher verlassen kann [FKN<sup>+</sup>14]. Da dieses Orakel globale Informationen benötigt und somit schwierig zu implementieren ist, wird dieses Orakel meistens nicht verwendet und es werden lokalere Orakel vorgestellt [FKN<sup>+</sup>14].

$\mathcal{NID}$ : Bezeichnet das Orakel „no identifiers“, welches *true* ausgibt, wenn die ID vom angegebenen Knoten  $u$  nicht in einem Knoten  $v$  oder in einer Nachricht im Channel von  $v$  für einen *relevanten* Knoten  $v \neq u$  existiert [FKN<sup>+</sup>14]. Einfacher gesagt gibt das Orakel *true* aus, wenn der angegebene Knoten  $u$  keine eingehenden impliziten oder expliziten Kanten von einem Knoten besitzt, der nicht „tot“ oder „scheintot“ ist [Sch20].

$\mathcal{EC}$ : Bezeichnet das Orakel „empty channel“ und gibt *true* aus, wenn der Channel für den angegebenen Knoten  $u$  leer ist [FKN<sup>+</sup>14].

$\mathcal{NIDEC}$ : Ist ein Verbund aus  $\mathcal{NID}$  und  $\mathcal{EC}$  und gibt *true* aus, wenn beide Orakel wahr sind [FKN<sup>+</sup>14].

$\mathcal{ONESID}$ : Gibt *true* aus, wenn der angegebene Prozess  $u$  eine Kante zu mindestens einen *relevanten* Knoten  $v$  besitzt [FKN<sup>+</sup>14].

Foreback et al. zeigen zudem, dass es keine selbststabilisierende Lösung für das  $\mathcal{FDP}$  gibt, welche nicht von einem *id-sensitive* Orakel abhängt. Wir werden jedoch später sehen, dass das Relay-Modell, welches nachfolgend erläutert wird, mit einem anderen Ansatz das  $\mathcal{FDP}$  lösen kann, ohne ein außenstehendes Orakel nutzen zu müssen.



## 2.4 Relay-Modell

Wie in Abschnitt 2.3.1 beschrieben, kann es kein selbststabilisierendes Protokoll geben, welches keine Orakel benötigt. Ein anderen Ansatz für das Verbinden von Knoten stellen jedoch Scheideler und Setzer [SS18] vor. Dieser soll das Lösen des  $\mathcal{FDP}$  in selbststabilisierenden Protokollen ohne Orakel ermöglichen. Das vorgestellte Modell namens Relay-Modell verwendet als Grundlage eine zuverlässige Verbindungsschicht oder Link-Layer genannt und für die Verbindung zu anderen Knoten sogenannte Relays [SS18, Set20]. Ein Relay dient dabei als Verbindungsschnittstelle zwischen zwei Knoten. Relays können verglichen werden mit Sockets, die von einem Relay-Layer verwaltet und kontrolliert werden. Außerdem können die Verbindungsinformationen zu einem Relay nicht weitergegeben werden, sondern müssen durch den entsprechenden Knoten autorisiert werden [SS18]. In diesem Relay werden weiterhin alle Informationen gespeichert, welche die eingehenden und ausgehenden Verbindungen ermöglichen. Die zwei folgenden grundlegende Probleme des Standardmodells von Overlay Netzwerken werden durch die Relays gelöst. Das erste Problem ist, dass Knoten in einem Netzwerk ohne Einschränkungen Referenzen von einem Knoten  $v$  an andere Knoten schicken können, wodurch der Knoten  $v$  lokal nicht entscheiden kann, ob dieser eingehende Verbindungen besitzt oder nicht [Set20]. Somit kann dieser Knoten nicht entscheiden, ob dieser das System verlassen kann ohne den schwachen Zusammenhang des Systems zu zerstören. Das zweite Problem besteht in der asynchronen Nachrichtenverarbeitung in den Netzwerken. Da die Knoten unterschiedlich lang für die Verarbeitung von Nachrichten benötigen dürfen, kann ein Knoten  $v$  nicht wissen, ob Nachrichten mit Information, welche wichtig für die Konnektivität des Systems sind, noch unterwegs zu  $v$  sind oder nicht [Set20]. Im Prinzip kann ein Knoten die entsprechenden Relays abfragen, ob diese Verbindungen eingehende Kanten besitzen. Zudem können Verbindungen über Relays nur geschlossen werden, wenn alle Nachrichten, welche die Verbindung betreffen, weitergeleitet und verarbeitet wurden [Set20]. Das Modell ist jedoch nicht nur eine Implementierung eines Orakels zum Lösen des  $\mathcal{FDP}$ , sondern besitzt folgende Vorteile:

1. Das Relay-Modell kann selbststabilisierend implementiert werden [Set20].
2. Das Modell ist universell, was bedeutet, dass beliebige initiale Systemgraphen in andere beliebige Graphen transformiert werden können [Set20].
3. Bereits existierende Protokolle können so verändert werden, dass diese mit dem Relay-Modell ausgeführt werden können [Set20].

Innerhalb dieser Arbeit wird später ein vorhandenes Protokoll umgebaut, sodass es mit der Implementierung des Relay-Modells simuliert werden kann.

Ein weiterer wichtiger Aspekt des Relay-Modells ist die Möglichkeit, lokal Rechte für Verbindungen entweder zu gewähren oder diese zu widerrufen [Set20]. Diese Eigenschaft nutzen wir später für die Erweiterung des Modells in Bezug auf der DoS Resistenz (Abschnitt 4).

### 2.4.1 Aufbau des Modells

Der grundlegende Aufbau des Relay-Modells besteht aus dem Relay-Layer, einem zuverlässigen Link-Layer und den einzelnen Relays. Alle Verbindungen zwischen Prozessen bzw. Knoten über Relays werden durch den Relay-Layer verwaltet [SS18, Set20]. Jeder Knoten besitzt einen eigenen Relay-Layer, wodurch generell ersichtlich ist, welcher Relay zu welchem Knoten gehört [SS18]. Dieser Relay-Layer wird nachfolgend  $RL(v)$  genannt, was den Relay-Layer von dem bestimmten Knoten  $v$  bezeichnet. Weiterhin soll der Relay-Layer auf der gleichen Maschine laufen

wie der Knoten selbst [SS18]. Grund dafür ist, dass die Kommunikation zwischen Knoten und Relay-Layer lokal durchgeführt werden soll [SS18]. Weiterhin besitzt jeder Relay-Layer eine eindeutige ID, welche nachfolgend *RID* genannt wird [SS18]. Diese ist abhängig von der Adresse der Maschine, sodass Nachrichten an den entsprechenden Endpunkt gesendet werden können, wenn man die *RID* des Relay-Layers kennt [SS18].

Der Link-Layer des Modells ist für die Weiterleitung von Nachrichten verantwortlich. Um das umzusetzen, überwacht der Link-Layer bestimmte Buffer, welche alle Nachrichten enthalten, die verarbeitet werden sollen [SS18]. Für interne Nachrichten zwischen den Relay-Layern, welche zum Beispiel genutzt werden, um einen anderen Relay-Layer darüber zu informieren, dass ein bestimmtes Relay geschlossen wurde, besitzt jeder Relay-Layer einen Buffer  $RL(v).Buf$  [SS18]. Im originalen Protokoll werden diese Buffer mit Tupeln befüllt, welche als ersten Eintrag die *RID* des Relay-Layers, der die Nachricht empfangen soll, besitzen [SS18]. Im zweiten Eintrag des Tupels steht die Nachricht, die übermittelt werden soll [SS18]. Weiterhin besitzt jedes Relay einen solchen Buffer  $r.Buf$ , der vom Link-Layer überwacht wird [SS18]. Der Relay-Layer darf jedoch in beide Arten von Buffern nur Nachrichten einfügen [SS18]. Lediglich der Link-Layer ist dazu berechtigt Nachrichten aus diesen Buffern zu entfernen, sobald er diese verarbeitet hat [SS18].

Die Informationen für Verbindungen zu anderen Knoten werden alle in den zugehörigen Relays gespeichert, die durch den  $RL(v)$  verwaltet werden [SS18]. Jedes Relay besitzt eine global eindeutige *ID*, welche die *RID* des Relay-Layers beinhaltet, und den Status *state*, welcher angibt, ob der Relay geschlossen oder aktiv ist [SS18]. Die *RID* kann mit der Schreibweise  $RID(r)$  aus der ID des Relays  $r$  entnommen werden [SS18]. Jedes Relay kann entweder genau eine ausgehende Verbindung besitzen oder keine [SS18]. Diese Information wird in dem *out* Attribut innerhalb eines Tupels gespeichert [SS18]. Dabei ist der erste Eintrag ein Set von Keys und der zweite die ID des Relays, zu dem die ausgehende Verbindung gerichtet ist [SS18]. Sollte ein Relay keine ausgehende Verbindung zu einem anderen Knoten besitzen wird dieser als Sink-Relay bezeichnet [SS18]. Sollten Nachrichten an einem Sink-Relay  $r$  ankommen, werden diese an den Knoten  $v$  des  $RL(v)$ , welcher den Relay  $r$  verwaltet, gesendet [SS18]. Ein Relay kann, im Gegensatz zu den ausgehenden Verbindungen, mehrere eingehende Verbindungen besitzen [SS18]. Diese Verbindungen werden als Tripel entweder in der Form  $(key, RID, \perp)$  oder  $(key, \perp, r)$  in dem Set innerhalb des *In* Attributs gespeichert [SS18]. Die *RID* in der ersten Form gibt an, welcher Relay-Layer mit dem entsprechenden Key eine Nachricht an dieses Relay senden darf [SS18]. Die zweite Form beinhaltet das Relay, über das eine Referenz des Relays, welche diese eingehende Verbindung besitzt, weitergeleitet wurde [SS18]. Diese Verbindung ist noch nicht bestätigt und muss erst durch Nachrichten aktiviert werden [SS18]. Sollten nicht bestätigte Verbindungen mit dem entsprechenden Key später aktiviert werden, wird die eingehende Verbindung von der zweiten Form in die erste Form gebracht [SS18]. Da Relays Ketten bilden können, mit denen Nachrichten über mehrere Knoten an ein bestimmtes Ziel weitergeleitet werden, brauchen wir Attribute, die Informationen über diese Ketten zur Verfügung stellen. Das erste Attribut ist das Level  $r.level$ , welches angibt, wie viele Relays ab dem Relay  $r$  in der Kette folgen, bis ein Sink-Relay erreicht wird [SS18]. Innerhalb des Attributs  $r.sinkRID$  wird dann die *RID* des Relay-Layers gespeichert, welcher das Sink-Relay der Kette, in der sich  $r$  befindet, verwaltet [SS18]. Zusätzlich besitzt jedes Relay, wie oben beschrieben, einen Buffer, in den Nachrichten eingefügt werden, wenn diese an den Relay gesendet werden [SS18].

Der Relay-Layer verwaltet eine Liste von Relays, welche die oben beschriebenen Eigenschaften besitzen. Zusätzlich zu dem Verwalten der Relays können die folgenden Funktionen auf dem Relay-Layer ausgeführt werden.

- Mit **new Relay** kann eine neue Referenz eines Relays vom Relay-Layer abgerufen werden [SS18]. Dieses Relay ist standardmäßig ein Sink-Relay [SS18].

- Mit **delete**  $\hat{r}$  kann ein Relay gelöscht werden [SS18], wobei das Relay nicht direkt gelöscht, sondern als „zu Löschen“ markiert wird, damit der Relay-Layer mögliche Verbindungen korrigieren kann [SS18].
- Mit **getRelays** kann eine Liste von Relay-Referenzen, welche noch aktiv sind, abgerufen werden [SS18].
- Um Nachrichten über Relays senden zu können, wird vom Relay-Layer die Funktion **send**( $\hat{r}, action(parameters)$ ) bereitgestellt, wobei  $\hat{r}$  die Referenz eines Relays ist, über das die Aktion gesendet werden soll [SS18].
- Mit **merge**( $R$ ) können Mengen von Relay-Referenzen gemerged werden [SS18]. Die Funktion gibt entweder eine neue Relay-Referenz zurück, wenn das Mergen der Relays erfolgreich war, oder macht nichts, wenn die Relays keine gleichen Informationen speichern [SS18]. Für die genauen Bedingungen des Befehls wird auf die Referenzen [SS18] und [Set20] verwiesen.

Neben diesen Aktionen können über den Relay-Layer Informationen bezüglich des Status der Relay-Referenzen abgefragt werden. Dafür sind die Funktionen **incoming**( $\hat{r}$ ), **direct**( $\hat{r}$ ), **is-sink**( $\hat{r}$ ), **dead**( $\hat{r}$ ) und **same-target**( $\hat{r}_1, \hat{r}_2$ ) vorgesehen [SS18].

- **incoming** gibt die Anzahl der eingehenden Verbindungen eines Relays zurück [SS18].
- Mit **direct** kann geprüft werden, ob das Level des Relays kleiner gleich eins ist und somit der bestimmte Relay entweder ein Sink-Relay ist oder ein Relay, welches eine direkte Verbindung zu einem Sink-Relay besitzt [SS18].
- **is-sink** gibt, im Gegensatz zu der **direkt** Funktion, nur *true* zurück, wenn das Level des gegebenen Relays gleich 0 ist und der Relay somit ein Sink-Relay repräsentiert [SS18].
- **dead** gibt *true* aus, wenn der Status des angegebenen Relays auf *dead* steht und der Relay gelöscht wurde [SS18].
- Mithilfe der **same-target** Funktion kann abgefragt werden, ob die beiden angegebenen Relays die gleiche ausgehende Verbindung besitzen [SS18].

Wenn ein Knoten selbst stoppt und somit „tot“ ist, wird der Relay-Layer über die Funktion *stop* auch gestoppt. Das bedeutet, dass der Relay-Layer alle Sink-Relays endgültig löscht und danach Relays, welche keine eingehenden Verbindungen mehr besitzen, nach und nach löscht [SS18]. Der Relay-Layer hört erst auf zu existieren, wenn alle Relays innerhalb des Relay-Layers gelöscht wurden [SS18].

In Abbildung 2.3 wurde eine mögliche Konfiguration eines Systems, welches das Relay-Modell benutzt, veranschaulicht. Dabei stellen die Kreise inklusive der Zahlen die Knoten dar. Die roten Rechtecke repräsentieren hingegen die Relays. Eine Verbindung von Knoten zu Relay bedeutet, dass der Relay-Layer des Knotens dieses Relay verwaltet. Die Sink-Relays werden durch ein „s“ gekennzeichnet. Die roten Verbindungen zeigen Verbindungen zwischen zwei Relays. Somit sind die Relays von Knoten 1 und 2 beide Relays mit Level 2 und besitzen keine direkte Verbindung zu dem Knoten 4. Jede Nachricht, die von Knoten 1 und 2 über diese Relays geschickt wird, wird zuerst über das Relay von dem Knoten 3 geschickt. Das Relay von Knoten 3, welches eine Verbindung zu dem Knoten 4 besitzt, ist hingegen ein direktes Relay. Genauso ist das Relay von Knoten 4, welches eine Verbindung zu dem Sink-Relay von dem Knoten 3 besitzt, auch ein direktes Relay. Dieses Relay besitzt aber keine eingehenden Verbindungen und ist somit nicht in einer Relay Kette. Weiterhin kann man erkennen, dass ein Relay mehrere eingehende Verbindungen haben kann aber nur eine ausgehende Verbindung. Zudem ist jedes Relay, was keine ausgehende Verbindung besitzt, automatisch ein Sink-Relay.

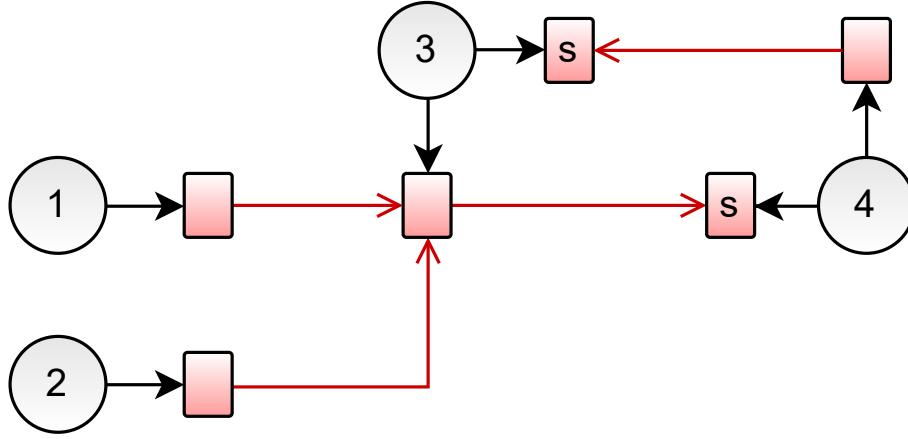


Abbildung 2.3: Möglicher Aufbau eines Systems mit dem Relay-Modell

### 2.4.2 Standardaktionen und Nachrichtenverarbeitung

Wie bereits in Abschnitt 2.1 beschrieben, sind auch im Relay-Modell Nachrichten, welche an andere Knoten gesendet werden, Aufrufe von Methoden in der Form  $action(parameters)$  [SS18]. Die Parameter dürfen Relay-Referenzen beinhalten, um zum Beispiel ein eigenes Sink-Relay an einen anderen Knoten zu senden, damit Verbindungen zum eigenen Knoten aufgebaut werden können. Da jedoch Relays nicht mit anderen Knoten geteilt werden dürfen, müssen Relay-Referenzen vor dem Senden der Aktion durch Informationen ausgetauscht werden, mit denen ein Knoten, der die Nachricht bekommt, eine gültige Verbindung aufbauen kann [SS18]. Dafür definieren Scheideler et al. [SS18] die Relay-Parameter, welche mit einem Quadruple in der Form  $(key, r.ID, r.level + 1, r.sinkRID)$  dargestellt werden, wobei  $r$  das Relay ist, welches in den Parametern als Referenz vorkommt und durch den Relay-Parameter ersetzt wird. Weiterhin muss eine neue eingehende Verbindung zu dem Relay  $r$ , welches gesendet wird, ermöglicht werden. Dafür wird beim Ersetzen der Relay-Referenz ein neuer  $key$  erstellt, welcher auch in dem Quadruple an erste Stelle steht [SS18]. Anschließend wird ein neues Tripel  $(key, \perp, s)$  in die eingehenden Referenzen von  $r$  eingefügt [SS18]. Das Relay  $s$  in der eingehenden Referenz ist das Relay über das die Aktion gesendet wird. Wenn solche Aktionen vom Link-Layer in den Buffer von Relays erfasst werden, werden diese an den Knoten gesendet, damit dieser die Aktion ausführen kann [SS18]. Zudem existieren weitere Standardaktionen, welche für den Relay-Layer reserviert sind. Darunter zählt die *TRANSMIT* Aktion, welche eine Nachricht  $m = ((key, inID, outID), action(parameters))$  als Parameter beinhaltet [SS18]. Die *TRANSMIT* Aktion wird dann benutzt, wenn ein Relay eine ausgehende Verbindung besitzt und die enthaltene Aktion weitergeleitet werden muss. Der erste Teil der Nachricht wird als Header bezeichnet [SS18]. Wenn ein Relay-Layer eine *TRANSMIT* Aktion erhält, wird zuerst geprüft, ob der Relay-Layer ein Relay besitzt, für das der Header gültig ist [SS18]. Sollte der Header für kein Relay gültig sein, wird die Nachricht nicht verarbeitet und der Relay-Layer, von dem die Nachricht empfangen wurde, wird darüber informiert, warum die Nachricht nicht verarbeitet wurde [SS18]. Ein gültiger Header kann mit der nachfolgenden Definition von Scheideler et al. [SS18] geprüft werden:

**Definition 2.8** (Valider Nachrichten Header). *Eine Nachricht  $m = ((key, inID, outID), action(parameters))$  besitzt einen validen Header für ein Relay  $r$  genau dann, wenn  $r.ID = outID$  und entweder  $(key, RID, \perp) \in r.In$  mit  $RID = RID(inID)$ , oder  $(key, \perp, r') \in r.In$  mit  $r'.sinkRID = RID(inID)$*

Weitere Aktionen sind die *NOT\_AUTHORIZED* und die *OUT\_RELAY\_CLOSED* Akti-

on. Beide Aktionen werden an andere Relay-Layer gesendet, wenn der Header einer Transmitt Nachricht nicht valide ist [SS18]. Die *NOT\_AUTHORIZED* Aktion wird gesendet, wenn ein Relay für den Header gefunden wurde, dieser jedoch nicht den Key des Headers in einer seiner eingehenden Referenzen besitzt [SS18]. Die *OUT\_RELAY\_CLOSED* Aktion wird gesendet, wenn kein Relay für den Header einer Transmittnachricht gefunden wurde [SS18]. Außerdem wird beim Löschen eines Relays über die **delete** Funktion für alle eingehenden Referenzen eine *OUT\_RELAY\_CLOSED* Aktion an die entsprechenden Relay-Layer gesendet [SS18]. Weiterhin führt der Relay-Layer periodisch eine Timeout Funktion aus, welche alle Relays und deren Informationen korrigiert und gegebenenfalls andere Relay-Layer über Korrekturen informiert [SS18]. Um dies zu gewährleisten, werden für jede explizite eingehende Referenz in jedem Relay eine *PING* Aktion an den bestimmten Relay-Layer gesendet [SS18]. Beim Erhalt dieser *PING* Nachricht prüft der Relay-Layer, ob dieser ein Relay besitzt, der zu den angegebenen Daten passt, und passt gegebenenfalls Level Informationen an [SS18]. Sollte der Relay-Layer kein Relay für die *PING* Aktion besitzen, wird eine *IN\_RELAY\_CLOSED* Aktion an den Relay-Layer, von dem die *PING* Aktion empfangen wurde, gesendet [SS18]. Beim Erhalt der *IN\_RELAY\_CLOSED* Aktion kann der Relay-Layer alle eingehenden Referenzen seiner Relays anhand der übergebenen Keys und IDs korrigieren und löschen [SS18]. Des Weiteren werden in der Timeout Funktion *PROBE* Aktionen versendet, welche die impliziten eingehenden Referenzen prüfen sollen [SS18]. Die *PROBE* Aktionen werden innerhalb der *TRANSMIT* Verarbeitung bearbeitet und es wird geprüft, ob der Relay-Layer Relays besitzt, welche die zu prüfenden Keys in den ausgehenden Referenzen besitzen [SS18]. Sollte der Relay-Layer für einen Key innerhalb der *PROBE* Aktion kein Relay besitzen, wird eine *PROBEFAIL* Aktion an den betreffenden Relay-Layer gesendet, damit dieser die falsche Information korrigieren kann [SS18]. Beim Erhalt der *PROBEFAIL* Aktion wird entweder die eigene Relay Information angepasst oder, wenn es sich um eine Kette von Relays handelt, diese Aktion weiter zurückgeschickt [SS18]. Dafür wird innerhalb der *PROBE* Aktion die Key Sequenz der Kette mit übergeben [SS18]. Zuletzt muss noch erwähnt werden, dass, wenn ein Relay-Parameter in einer Aktion innerhalb einer Transmittnachricht an einem Sink-Relay ankommt, der Relay-Layer ein neues Relay mit den Informationen aus dem Relay-Parameter erstellt und den Parameter der Aktion dann durch die neue Relay-Referenz ersetzt [SS18].

Für den kompletten Ablauf und Pseudocode der gesamten Aktionen wird auf die Referenzen [SS18] und [Set20] verwiesen.

### 2.4.3 *FDP* Lösung durch das Relay-Modell

Da das Relay-Modell entwickelt wurde, um das *FDP* zu lösen, stellte Setzer 2020 [Set20] ein Framework vor, welches auf selbststabilisierende Protokolle angewandt werden kann.

Das Protokoll, welches für das Relay-Modell umgeändert wird, soll die nachfolgenden drei Annahmen erfüllen. Die erste Annahme ist die **Single-sink Annahme**, welche besagt, dass jeder Knoten  $u$  im System nur genau ein Sink-Relay besitzen soll, welches auch das einzige Relay sein soll, das eingehende Verbindungen besitzt [Set20]. Dieses Sink-Relay wird innerhalb des Knotens als  $u.in$  gespeichert [Set20]. Mit der **Self-introduction Annahme** soll jeder Knoten  $u$  regelmäßig in seiner Timeout Funktion seinen Sink-Relay über jedes direkte Relay in  $u$  senden und sich somit selbst vorstellen [Set20]. Die grundlegende Topologie besitzt, anders als das Relay-Modell erlaubt, keine indirekten Verbindungen zu Knoten über andere Knoten [Set20]. Dadurch werden indirekte Relays umgekehrt, sobald diese auftauchen oder existieren [Set20]. Diese Annahme heißt **Direct relay Annahme**.

Das Framework definiert dafür eigene Aktionen für Knoten und eine eigene Timeout Aktion [Set20]. In der Timeout Aktion des Frameworks kann jedoch die Timeout Aktion des zugrundeliegenden Protokolls ausgeführt werden [Set20]. Weiterhin führt das Framework ei-

ne  $REPLACE_A()$  Aktion aus, wenn eine Aktion des zugrundeliegenden Protokoll ausgeführt werden soll [Set20]. In dieser Aktion wird entschieden, ob die originale Aktion ausgeführt werden soll oder nicht [Set20]. Dadurch können, wenn ein Knoten das System verlassen möchte, Aktionsaufrufe verhindert werden, sodass das Verlassen gesichert wird. Der allgemeine Ablauf des Frameworks läuft wie folgt ab:

- Periodisch werden alle indirekten Relays innerhalb des Knotens, welche keine eingehenden Verbindungen besitzen, mittels Relay Umkehrung gelöscht [Set20].
- Jeder Knoten  $u$ , der das System verlassen möchte, sucht sich einen Knoten, welcher im System bleibt [Set20]. Dieser Knoten wird Anker genannt und ist für die Umkehrung von Relays verantwortlich, welche  $u$  entfernen möchte [Set20].
- Jeder Knoten  $u$ , der das System verlassen möchte, versucht alle Relays, welche keine Sink-Relays sind, zu entfernen [Set20]. Der Knoten führt eine Relay Umkehrung aus, wenn dieses Relay keine eingehenden Verbindungen mehr besitzt [Set20].
- Jeder Knoten  $u$ , der das System verlassen möchte, entfernt alle eingehenden Verbindungen [Set20]. Durch die **Self-introduction Annahme** erreicht den Knoten  $u$  periodisch eine Vorstellungsnachricht über diese eingehenden Verbindungen [Set20]. Beim Erhalt dieser Nachricht kann nun der Knoten den sendenden Knoten anfragen, diese Verbindung umzukehren, sodass die eingehende Verbindung zu Knoten  $u$  entfernt wird [Set20]. Weiterhin baut der Knoten  $u$  keine weiteren Verbindungen zu anderen Knoten auf [Set20].

Mit diesem Ablauf besitzen Knoten, die das System verlassen wollen, nach endlicher Zeit keine Relays mit eingehenden Verbindungen und ein Relay mit einer ausgehenden Verbindung zu dem zuvor ermittelten Anker [Set20]. Somit kann der Knoten in diesem Fall das System sicher verlassen [Set20].

Setzer beweist dafür den folgenden Satz [Set20]:

**Satz 2.9.** *In jeder Berechnung des zugrundeliegenden Protokolls bleibt jeder bleibende Knoten weiterhin aktiv und jeder Knoten, der das System verlassen möchte, wird irgendwann inaktiv, wenn mindestens ein bleibender Knoten vorhanden ist.*

Dadurch ist gegeben, dass alle verlassenden Knoten irgendwann inaktiv sind und aus dem System austreten können.

Für die genauen Erklärungen der Aktionen, welche durch das Framework hinzugefügt werden, deren Ablauf und Pseudocode wird auf die Referenz [Set20] verwiesen.

## 2.5 Denial of Service

Ein Teil dieser Arbeit wird es sein, die Implementierung des Relay-Modells in der Hinsicht zu erweitern, dass DoS Angriffe erkannt und gegebenenfalls verhindert werden können. Bevor wir jedoch die Erweiterungen vorstellen können, müssen wir zuerst definieren, was mit dem Begriff Denial of Service gemeint ist.

Ein Denial of Service Angriff kann erstmal beschrieben werden als Angriff, welcher Computer oder Netzwerke davon abhält seine normalen Dienste zur Verfügung zu stellen und diese zu erreichen [DM03, CKBR06]. Ein DoS Angriff besteht nicht aus zufälligen Ausfällen, sondern entsteht nur durch einen absichtlich bösartigen Angriff, welcher Ressourcen blockiert oder schwächt [DM03]. Die Einschränkung des Angriffes muss nicht permanent gegeben sein, sondern kann auch temporär die Ressource unerreichbar machen [DM03]. Douligieris et al. [DM03] differenzieren hierbei die folgenden DoS Arten. Bei den DoS Angriffen in dem Netzwerk Gerät

Level, werden Fehler in der Software eines Netzwerkgeräts ausgenutzt oder es wird versucht die Hardware dieses Geräts zu überlasten [DM03]. In der Betriebssystem-Ebene, werden Implementierungen von Protokollen innerhalb eines Betriebssystems ausgenutzt, um einen DoS Angriff durchzuführen [DM03]. DoS Angriffe, welche sich in der Applikationsebene bewegen, versuchen Fehler innerhalb eines Programms, das auf dem angegriffenen System läuft, auszunutzen. Alternativ wird versucht, Ressourcen des Systems mithilfe dieses Programms zu blockieren [DM03]. Die Data Flooding Kategorie bezeichnet DoS Angriffe, welche keine speziellen Fehler ausnutzen, sondern die Bandbreite des Systems benutzen, um ein bestimmtes Ziel mit möglichst vielen Daten zu fluten, damit dieses überfordert ist und somit eingeschränkt wird [DM03]. In der letzten Kategorie bewegen sich Angriffe, welche bestimmte Eigenschaften von Standardprotokollen verwenden, um Systeme einzuschränken [DM03].

Eine weitere Unterkategorie von DoS Angriffen, sind die Distributed Denial of Service Attacks oder kurz: DDoS Angriffe. Bei diesen Angriffen, werden DoS Angriffe über sogenannte Zombies ausgeführt, welche von einem Angreifer übernommene Systeme darstellen [DM03, CKBR06, GJM12, LRST00]. Der eigentliche Angriff wird über den Master durchgeführt, welcher den Angriff und die Zombies koordiniert [LRST00].

In dieser Arbeit betrachten wir jedoch nur DDoS Angriffe, bei denen von vielen unterschiedlichen Knoten Nachrichten an einen speziellen Knoten gesendet werden.

Eine mögliche Methode zur Erkennung von DDoS Angriffen bildet die Erkennung von Anomalien innerhalb des Systems [DM03]. Um das zu erkennen, wird das System und sein Verhalten auf nicht normale Verhaltensweisen überprüft [DM03]. Zugrundeliegend besteht dafür immer ein normales Systemverhalten von dem abgewichen wird [DM03]. So ein abnormales Verhalten kann auftauchen, wenn viele Systeme zu einem bestimmten Zeitpunkt gleichzeitig Nachrichten an ein anderes System senden und somit die Nachrichtenmenge bei dem System abnormal hoch ist. Carl et al. [CKBR06] definieren dafür die Change-point Erkennung, bei der durch Angriffe resultierende statistische Änderungen des Netzwerkverkehrs erkannt werden. Für die Durchführung der Erkennung schlagen Carl et al. den CUSUM Algorithmus vor. In dieser Arbeit werden wir jedoch eine Schiebefenstermethode verwenden. Wie später in Abschnitt 4 erklärt, basiert dieser Algorithmus auf der Idee von Kifer et al. [KBG04].

Das Relay-Modell bietet zudem einige Eigenschaften, welche DoS Angriffe erschweren können. Dadurch, dass Verbindungen von Systemen über Relays durchgeführt werden, kann jeder Relay-Layer selbst entscheiden, welches System eingehende Verbindungen zum eigenen Relay-Layer besitzen darf und welches nicht. Außerdem können Verbindungen, welche aus Verkettungen von Relays bestehen, durch ein Relay-Layer innerhalb der Kette unterbrochen werden. Somit können Nachrichten das Sink-Relay der Kette nicht mehr erreichen. Sollte also ein Relay-Layer innerhalb dieser Kette einen Angriff erkennen, kann dieser die Verbindung direkt entfernen und so den Angriff auf den Knoten, der das Sink-Relay der Kette verwaltet, verhindern.





# Implementierung des Relay-Modells

Der wesentliche Bestandteil dieser Arbeit ist es, das Relay-Modell als nutzbare Library zu implementieren. Nachfolgend werden die wichtigsten Klassen der implementierten Library vorgestellt. Die Library wurde in Python programmiert und benötigt mindestens Python 3.6 oder eine höhere Version. Zusätzlich wird das Pip Modul `zmq` beziehungsweise `pyzmq` benötigt, welches die Python Implementierung der ZeroMQ Library ist. Alle Klassen befinden sich in dem Modul `RelayModel` und dann in den entsprechenden Dateien. Die genauen Beschreibungen der einzelnen Attribute und Funktionen sind aus den Dokumentationen des Moduls zu entnehmen. Nachfolgend wird nur der grobe Aufbau der Implementierung und die grundlegenden Änderungen zu dem eigentlichen Relay-Modell erläutert. Wenn nachfolgend von `LinkLayer` und `RelayLayer` gesprochen wird, wird damit die implementierte Klasse gemeint.

## 3.1 Modul Konfiguration

Grundlegende Parameter, die für die gesamte Library gelten, werden in der `ModuleConfig.py` Datei festgehalten. Folgende Parameter können dort angepasst werden:

- `RELAY_LOG_LEVEL`: Gibt an, welche Nachrichten in die Log Dateien geschrieben werden. Standardmäßig steht der Wert auf `logging.WARNING`, wodurch nur schwerwiegende Nachrichten, wie Warnungen und Fehler niedergeschrieben werden.
- `RELAY_LAYER_TIMEOUT_PERIOD`: Setzt die Zeit in Sekunden, welche zwischen zwei Timeouts des `RelayLayers` gewartet wird. Standardmäßig ist dieser Wert auf 1 Sekunde gesetzt.
- `NODE_TIMEOUT_PERIOD`: Setzt die Zeit in Sekunden, welche zwischen zwei Timeouts der Node Klasse gewartet wird. Standardmäßig ist dieser Wert auf 1 Sekunde gesetzt.
- `POLL_TIMEOUT`: Die Poll Timeout Zeit gibt in Millisekunden an, wie lange der `LinkLayer` auf eine Antwort nach dem Senden einer Nachricht wartet, bevor es erneut probiert wird. Das genaue Vorgehen wird in 3.6.1 näher beschrieben.
- `POLL_TRIES`: Der Parameter gibt an, wie oft der `LinkLayer` versucht eine Nachricht zu senden. Wenn die Versuche überschritten wurden, wird die Nachricht verworfen.

- **CONSIDER\_AS\_CLOSED**: Gibt eine Flag an, mit der entschieden wird, ob eine Closed Nachricht an den Knoten gesendet wird, wenn ein **LinkLayer** nicht erreichbar ist. Das genau Vorgehen des Fehlermanagements wird in 3.6.2 beschrieben.
- **STATE\_MONITOR\_PORT**: Falls die Library im Analysemodus ausgeführt wird, wird hiermit angegeben, unter welchem Port der StateMonitor zu erreichen ist. Der StateMonitor wird im Abschnitt 5.1 näher beschrieben.
- **STATE\_MONITOR\_ADDRESS**: Speichert die IP unter der der StateMonitor erreichbar ist, sollte der Analysemodus eingeschaltet sein.
- **DOS\_DETECTION\_ACTIVATED**: Mit dieser Variable kann die Erkennung von DoS Angriffen (siehe Abschnitt 4) ein- und ausgeschaltet werden. Standardmäßig ist dieser Wert auf *True* gesetzt und somit aktiv.
- **WINDOW\_SIZE**: Gibt an wie groß das Schiebefenster, welches für die Erkennung von DoS Attacken genutzt wird, pro Relay sein soll. Standardwert ist hierbei 20.
- **CHANGE\_ALPHA**: Der Schwellenwert für die Erkennung von DoS Attacken wird hiermit gesetzt. Dieser sollte grundlegend negativ sein, da sonst fallende Änderungen der Rate erkannt werden. Standardmäßig steht dieser auf -10.
- **NO\_MONITOR\_ACTIONS**: Aktionen, welche in dieser Liste stehen, werden nicht in die Erkennung von DoS Attacken mit einbezogen. Normalerweise handelt es sich hierbei um die Nachrichtentypen, welche nur von dem **RelayLayer** selbst verwendet werden und somit für das System reserviert sind.
- **RESULTS\_FOLDER**: Gibt den Pfad an, in dem die Ergebnisdateien abgelegt werden sollen, welche von dem StateMonitor bei einer Analyse des Systems geschrieben werden.

## 3.2 Identifikation von Relays und dem RelayLayer

Zur Identifikation von Relays und dem **RelayLayer** werden, wie in Abschnitt 2.4.1 beschrieben, eine RID und eine Relay ID, welche die RID beinhaltet, benötigt.

### 3.2.1 RelayLayerId

Die RID eines **RelayLayers** wird mit der Klasse **RelayLayerId** aus der **RelayId.py** Datei abgebildet. Wie in Abbildung 3.1 zu sehen, besteht diese aus der IP, welche den Rechner definiert auf dem der **RelayLayer** läuft, und dem Port unter dem dieser erreichbar ist. Zusätzlich kann noch eine Node ID gesetzt werden, um eine Topologie aufbauen zu können.

### 3.2.2 RelayId

Weiterführend besitzt jedes Relay eine eindeutige ID, welche ein bestimmtes Relay genau beschreibt. Das originale Protokoll erfordert zudem, dass die ID eines Relays die RID des **RelayLayers** beinhaltet. Grund dafür ist, dass geprüft werden kann, zu welchem **RelayLayer** das gegebene Relay gehört. Dadurch können Nachrichten an den entsprechenden **RelayLayer** gesendet werden, wenn wir nur die ID des Relays besitzen.

Aufgebaut ist die **RelayId**, wie in Abbildung 3.1 gezeigt, aus der **RelayLayerId** des zugehörigen **RelayLayer**, welcher das Relay verwaltet, und einer Ganzzahl, welche ein bestimmtes Relay innerhalb des **RelayLayers** definiert. Somit besitzt jedes Relay innerhalb des **RelayLayers** eine eindeutige Zahl, welche bei 0 startet und für jedes neu erstellte Relay hochgezählt wird.

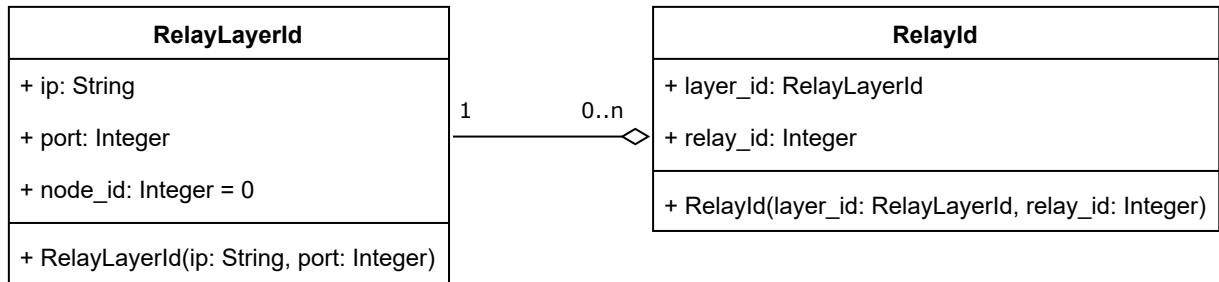


Abbildung 3.1: Klassendiagramm Relay Identifikation

### 3.3 Verbindungsreferenzen

Relays können, wie in Abschnitt 2.4.1 beschrieben, eingehende und ausgehende Kanten besitzen, um entweder Nachrichten zu empfangen, zu senden oder gegebenenfalls weiterzuleiten. Diese Verbindungen werden innerhalb eines Relays in den Variablen *r.out* und *r.In* gespeichert. Für die Implementierung dieser beiden Variablen werden nachfolgend zwei Klassen vorgestellt, welche die notwendigen Informationen speichern, sodass eingehende und ausgehende Verbindung nicht mehr in Tupeln und Tripel gespeichert werden müssen. Beide Klassen befinden sich innerhalb der *Relay.py* Datei.

#### 3.3.1 Ausgehende Referenzen

Die ausgehende Referenz eines Relays wird in der Klasse *OutReference* gespeichert. Alle Keys der ausgehenden Verbindung werden als String innerhalb eines Sets und die ausgehende ID mit der Klasse *RelayId* gespeichert. Auf die Keys und ID kann, wie in Abbildung 3.2 zu sehen, mit den Properties *keys* und *out\_id* der Klasse zugegriffen werden. Beim Instanzieren der Klasse können die *keys* und die *out\_id* direkt mit angegeben werden. Diese Angabe ist jedoch optional. Standardmäßig wird die *out\_id* auf *None* gesetzt und *keys* mit einem leeren Set initialisiert. Zusätzlich implementiert die Klasse *OutReference* die Methoden *add\_key* und *remove\_key*, um einzelne Keys aus dem Set von Keys zu entfernen und diese dem Set hinzuzufügen. Wenn die *out\_id* *None* ist, bedeutet dies, dass das Relay keine ausgehende Verbindung besitzt und das zugehörige Relay ein Sink-Relay ist.

#### 3.3.2 Eingehende Referenzen

Ebenso werden die eingehenden Referenzen, welche normalerweise als Tripel dargestellt werden, durch die Klasse *InReference* ersetzt. Wie in Abbildung 3.2 zu sehen, besitzt die Klasse die Attribute *key*, *rid* und *relay*, wobei *key* als String, *rid* als *RelayLayerId* Objekt und *relay* als *Relay* Objekt gespeichert wird. Wie beim Instanzieren der *OutReference* Klasse, ist es auch bei der *InReference* Klasse möglich, direkt die Parameter zu übergeben. Falls *key*, *rid* und *relay* nicht direkt übergeben werden, wird *key* als leerer String, *rid* als *None* und *relay* als *None* initialisiert. Weiterhin kann mithilfe der *check\_valid* Methode geprüft werden, ob die Referenz eine valide Referenz darstellt. Nicht valide ist eine eingehende Referenz, wenn der *key* leer ist, aber *rid* oder *relay* gesetzt ist [SS18].

### 3.4 Relay

Da der *RelayLayer* für jedes Relay bestimmte Informationen speichern muss, bietet es sich an alle Informationen in einer Klasse zu speichern. Diese Klasse ist die *Relay* Klasse, welche sich in

der `Relay.py` Datei befindet. Zu den eigentlichen Informationen, welche für das Protokoll genutzt werden, implementiert die `Relay` Klasse noch weitere Hilfsmethoden, um die Handhabung der eingehenden Verbindungen zu vereinfachen. Wie in Abbildung 3.2 zusehen, speichert die `Relay` Klasse alle wesentlichen Informationen für ein Relay innerhalb des `RelayLayers`. Jedoch wurde das `r.state` Attribut eines Relays zu `Relay.alive` geändert. Normalerweise wird der Status mit `dead` oder `alive` gespeichert. In dieser Ausarbeitung wurde der Status umgeändert zu `Relay.alive`, um die weitere Bearbeitung zu vereinfachen. Dabei ist `True` innerhalb des Attributs äquivalent zu dem originalen Status `alive` und `False` äquivalent zu dem Status `dead`. Wie in Abschnitt 2.4.1 beschrieben, soll der `RelayLayer` für jedes Relay einen Buffer speichern, worüber Nachrichten an den `LinkLayer` geschickt werden können. Da, wie später in 3.6.5 erläutert, die Buffer innerhalb des `LinkLayers` ausgelesen und verwaltet werden, werden innerhalb der `Relay` Klasse und des `RelayLayers` keine einzelnen Buffer gespeichert.

Zu den ursprünglichen Informationen eines Relays besitzt die `Relay` Klasse noch eine Flag innerhalb des `validated` Attributs. Diese Flag wird benötigt, um dynamisch erstellte Relays davor zu bewahren gelöscht zu werden, bevor diese überhaupt bearbeitet wurden. Dieser Fall entsteht, wenn eine Relay-Referenz innerhalb der Parameter einer Aktion an einen bestimmten Knoten gesendet wurde. Dabei erstellt der `RelayLayer` bei Erhalt der Transmittnachricht für den entsprechenden Relay-Parameter ein neues Relay und sendet die Aktion an den Knoten. Mit der Timeout Funktion des Knoten, in der alle Relays, welche sich nicht in Attributen des zugrundeliegenden Protokolls befinden, gelöscht werden, konnte es passieren, dass ein Relay gelöscht wurde, bevor die eigentliche Methode mit dem gelöschten Relay als Parameter ausgeführt werden konnte. Dafür wurde die Validierungsflag eingeführt, welche nur auf `True` gesetzt wird, wenn das Relay von dem Knoten verarbeitet wurde. Zudem wurde die Timeout Funktion eines Knoten so angepasst, dass dieser nur validierte Relays betrachtet. Standardmäßig ist der Wert der Validierungsflag beim Instanzieren eines Relayobjekts auf `True` gesetzt. Beim dynamischen Erstellen eines Relays durch Relay-Parameter wird die Flag jedoch erst auf `False` gesetzt.

Des Weiteren wurde in der `Relay` Klasse das Attribut `dos_threshold` hinzugefügt. Dies gibt die Grenze an, bei der durch die Denial Of Service Erkennung ein Angriff erkannt wird. Standardmäßig ist dieser Wert auf den Wert gesetzt, welcher in der Modulkonfiguration als `CHANGE_ALPHA` gesetzt wurde (siehe Abschnitt 3.1). Dieser Wert kann jedoch nach Belieben angepasst werden. Das genaue Verhalten der Erkennung und wie dieser Wert genutzt wird, wird später im Abschnitt 4 näher erklärt.

## 3.5 Kommunikation

Jegliche Kommunikation zwischen Nodes, `RelayLayer` und Relays geschieht über die Klassen innerhalb der `Communication.py` Datei.

Da das originale Relay-Layer Protokoll einen Austausch von Relay-Referenzen in Parametern mit Relay-Parametern erfordert (siehe Abschnitt 2.4.2), welche nur die wichtigen Informationen für die Verbindung zu einem Relay beinhalten, werden alle Relay-Referenzen anstatt durch eine Menge durch die Klasse Relay-Parameter ersetzt. Die Klasse enthält, wie in Abbildung 3.3 zu sehen, alle nötigen Informationen in den Attributen `key`, `relay_id`, `level` und `rid`, welche normalerweise über den Konstruktor gesetzt werden.

Weiterhin gibt es die Klassen `FailureMessage` und `SuccessMessage`, welche für den `LinkLayer` existieren. Sie geben an, ob eine Übertragung zu einem anderen `LinkLayer` erfolgreich war. Wann welche Nachricht übertragen wird, wird in 3.6.1 näher beschrieben.

Grundlegend arbeitet die Library mit der Aktionsklasse `Action`, welche in Wrapper Klassen eingebunden wird. Die verschiedenen Aktionsklassen sind nachfolgend in 3.5.1 beschrieben. Das

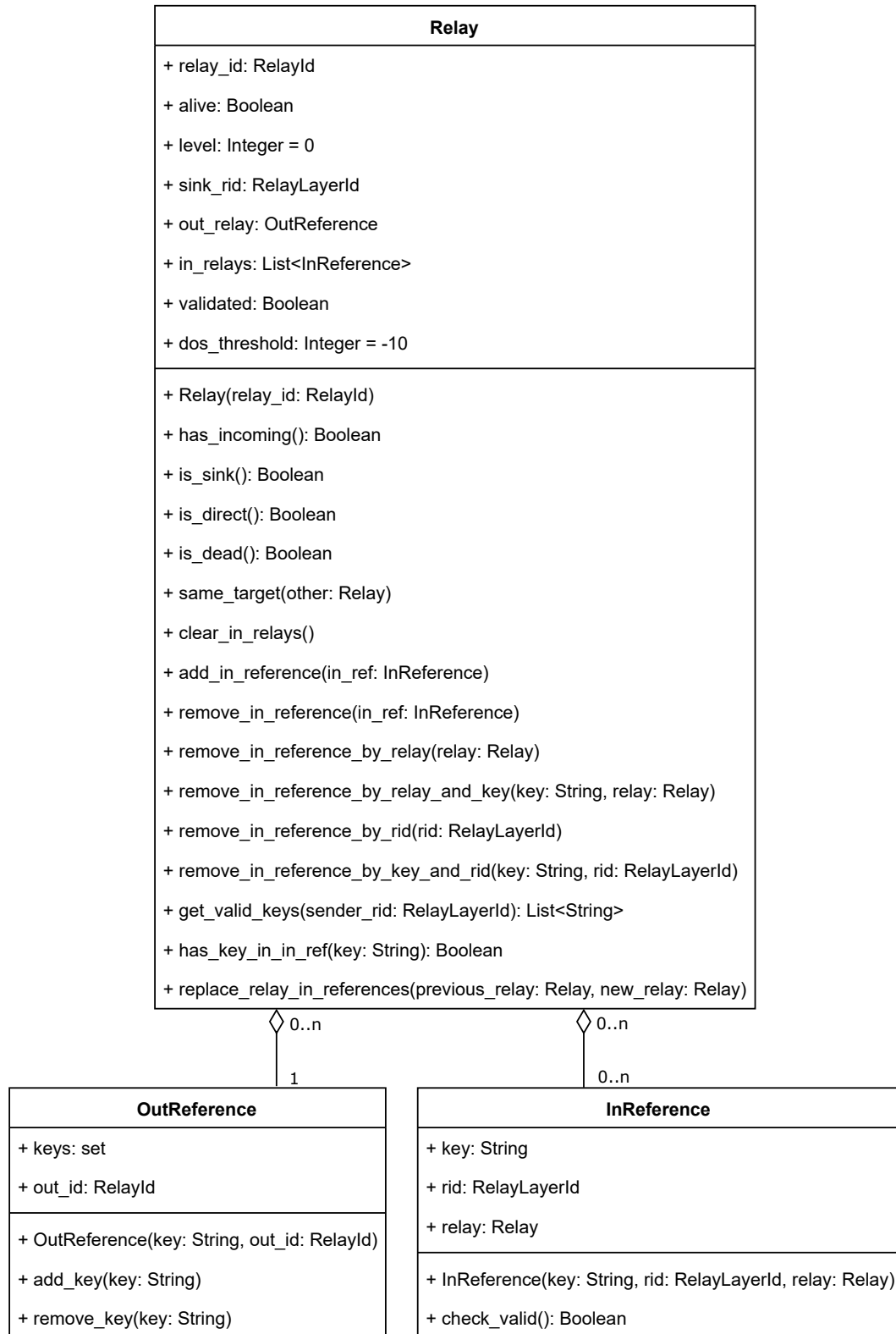


Abbildung 3.2: Klassen innerhalb der Relay.py Datei

originale Protokoll beschreibt zwei bestimmte Arten von Nachrichtenübertragungen. Die erste beinhaltet Nachrichten, die nur für **RelayLayer** bestimmt sind. Für die Übertragung wird

die `LayerMessage` Klasse verwendet. Sie beinhaltet die `RelayLayerId`, welche angibt an welchen **RelayLayer** die Nachricht gesendet werden soll, und die Action, welche ausgeführt werden soll sobald die Nachricht den entsprechenden **RelayLayer** erreicht hat. Der andere Nachrichtentyp wird dafür verwendet, jegliche Aktionen an einen entsprechenden Knoten zu senden. Diese Nachrichten werden mit der `TransmitMessage` Klasse repräsentiert. Innerhalb der `TransmitMessage` Klasse befindet sich ein Objekt der `Message` Klasse, welches wiederum eine Action und ein Header Objekt beinhaltet. Die Header Klasse speichert die Informationen, welche benötigt werden, um zu verifizieren, ob der **RelayLayer** ein Relay besitzt, über das eine Nachricht geschickt werden kann. Die Klasse besitzt ein Set aus Keys in dem `keys` Attribut, die Sender `RelayLayerId` in `sender_rid` und eine `RelayId` in dem `out_id` Attribut. All diese Klassen sind innerhalb der Abbildung 3.3 zu sehen, wobei die Details der Aktionsklassen separat dargestellt werden und in der Abbildung nur vereinfacht als Action Klasse abgebildet sind.

### 3.5.1 Aktionsklassen

Generell werden alle Nachrichten und Methodenaufrufe über die Action Klasse kommuniziert. Die Action Klasse enthält den Aktionstyp, welcher den Namen der Aktion definiert, und die Parameter der Aktion, welche standardmäßig als Liste gespeichert werden. Zudem wird in der Aktion das Relay gespeichert, welches die Aktion empfangen und an seinen Knoten weitergeleitet hat. Diese Speicherung wird für die Implementierung der Lösung für das *FDP* gebraucht (siehe Abschnitt 2.4.3).

Für jede Aktion des Relay-Layers, welche in Abschnitt 2.4.2 beschrieben wurden, gibt es eine eigene Klasse. Alle Aktionsklassen erben von der Action Klasse und setzen beim Erstellen der Klasse den `action_type` entsprechend dem Namen der Klasse. Alle Klassenvariablen in diesen Klassen werden nicht in den benannten Variablen gespeichert, sondern in der Parameterliste der Action Klasse. Zugriffen wird auf diese Variablen mithilfe von Properties, wodurch die entsprechenden Variablen in den Parametern bearbeitet werden. Somit ist es möglich die Klassenvariablen sowohl über den Namen der Variable als auch über die Parameter Liste zu holen und zu setzen.

Bei den Standard-Aktionsklassen handelt es sich, wie in Abbildung 3.4 zu sehen, um die `PingAction`, `ProbeAction`, `OutRelayClosedAction`, `InRelayClosedAction` und die `ProbeFailAction`. Eine Besonderheit besitzt jedoch die `PingAction` Klasse. Sie beinhaltet nicht die Attribute, welche für das Ausführen eines Pingbefehls benötigt werden, sondern eine Liste von Ping Objekten. Die `Ping` Klasse hingegen besitzt die Parameter, die dafür benötigt werden. Grundlegend für dieses Design ist die Beobachtung großer Mengen einzelner Ping Nachrichten an die gleichen Endpunkte, welche durch das beschriebene Design verhindert werden können. Somit wurden alle Pings, welche an den gleichen **RelayLayer** gesendet werden, in einer `PingAction` zusammengefasst. Dadurch wurde die Anzahl an Nachrichten über Sockets reduziert.

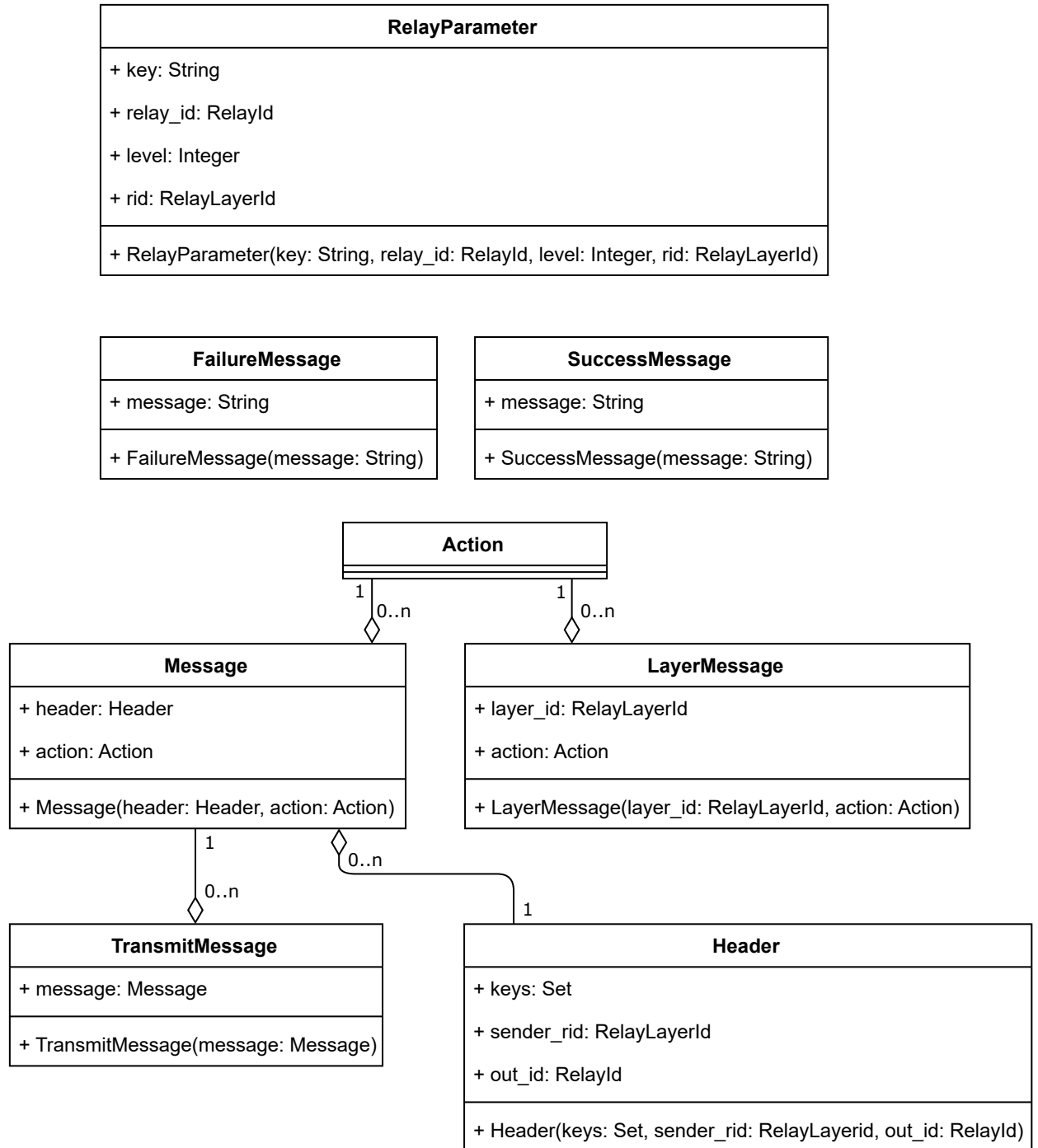


Abbildung 3.3: Klassen innerhalb der Communication.py Datei

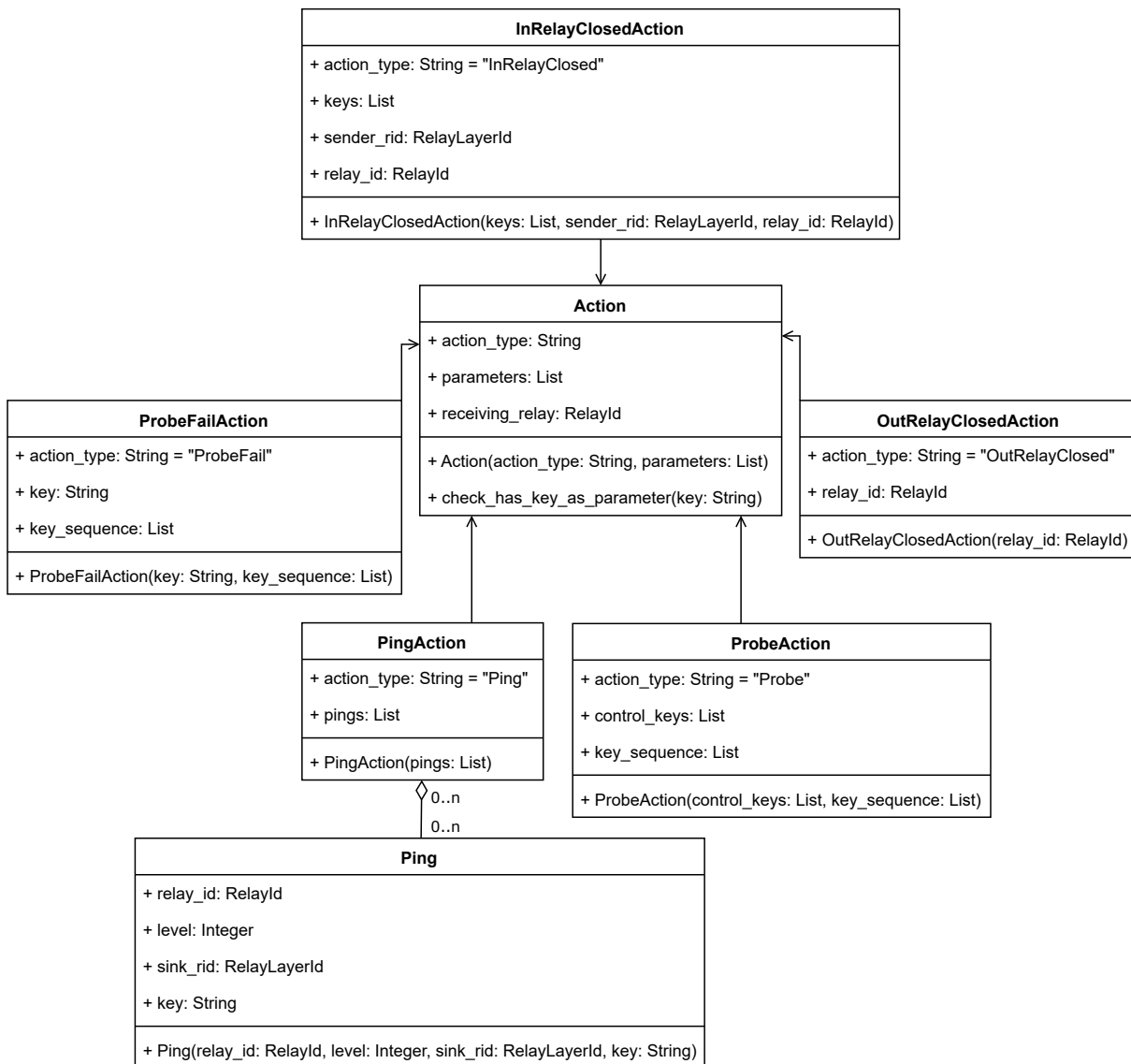


Abbildung 3.4: Klassendiagramm Actions



### 3.6 LinkLayer

Die **LinkLayer** Klasse, welche sich innerhalb der `LinkLayer.py` Datei befindet, ist für das Empfangen und Weiterleiten von Nachrichten an andere **LinkLayer** verantwortlich. Hauptsächlich überwacht die Klasse, wie es im originalen Protokoll gefordert wird, einerseits die Buffer der Relays, andererseits den Buffer des **RelayLayers** und sendet zudem enthaltene Nachrichten an die entsprechende Adresse. Weiterhin empfängt der **LinkLayer** eingehende Nachrichten und sendet diese an den **RelayLayer**. Generell sollte der **LinkLayer** aufgrund von Performancesteigerung innerhalb eines eigenen Prozesses gestartet werden. Dafür existiert die `start_link_layer` Methode innerhalb des `RelayModel/LinkLayer` Pakets. Die Methode erstellt ein **LinkLayer** Objekt und wartet so lange, bis der **LinkLayer** gestoppt wurde. Als Parameter der Funktion werden dieselben Parameter gefordert wie bei dem Konstruktor der Klasse selbst.

Generell besteht der **LinkLayer** aus drei Threads und zwei Prozessen. Der erste Thread ist der Worker-Thread, welcher überprüft, ob neue Nachrichten an den **LinkLayer** geschickt wurden, und ordnet diese dann in die entsprechenden Buffer ein. Der zweite Thread ist der Methodcalling-Thread, in dem die Schnittstelle zu dem **LinkLayer** Prozess überprüft wird. Diese Schnittstelle wird benötigt, um Funktionen innerhalb des **LinkLayers** auszuführen, obwohl das Objekt in einem anderen Prozess liegt. In diesem Thread werden alle Methodenaufrufe von anderen Prozessen, welche mit dem **LinkLayer** Prozess kommunizieren, entgegengenommen, bearbeitet und anschließend wird das Ergebnis der aufgerufenen Funktion zurückgegeben. Der letzte Thread ist dafür verantwortlich, die Relay Buffer zu verwalten und Nachrichten, welche in die Buffer eingefügt werden, an den richtigen **LinkLayer** zu schicken oder diese an den Knoten zu senden. Zusätzlich zu den Threads startet der **LinkLayer** zwei Prozesse innerhalb der Klasse. Der erste Prozess ist für das Empfangen von Nachrichten verantwortlich. Der Zweite Prozess wird erst gestartet, wenn ein **RelayLayer** registriert wird. Anschließend überwacht dieser den Buffer des **RelayLayers** und schickt die enthaltenen Nachrichten an andere **LinkLayer**. Der Grund für das Auslagern des Überwachens des **RelayLayer** Buffers in einen Prozess, wohingegen das Überwachen der Relay Buffer in einem Thread abläuft, wird in 3.6.5 näher begründet. Nachfolgend werden Pattern, Ausfallmanagement, wesentliche Funktionen und Designentscheidungen des **LinkLayers** näher erläutert.

#### 3.6.1 Verbindungen und Pattern

Der **LinkLayer** verwendet für die Verbindung zu anderen **LinkLayer** Sockets innerhalb der ZeroMQ Library. Jeder **LinkLayer** startet beim Erstellen einen Socket der Art *REP*, welcher nachfolgend Listeningsocket genannt wird, wodurch Nachrichten auf diesen Socket empfangen werden können. Andere **LinkLayer** können mit einem Socket der Art *REQ* eine Verbindung zu dem **LinkLayer** aufbauen und Nachrichten senden. Auf jede Nachricht, welche an dem Listeningsocket ankommt, folgt eine Antwort an den Sender. Je nachdem, ob die Nachricht dem richtigen Format entspricht, besteht diese Nachricht entweder aus einer *SuccessMessage* oder einer *FailureMessage*. Der **LinkLayer** akzeptiert nur Nachrichten, welche Objekte der *LayerMessage* und *TransmitMessage* Klasse sind. Alle anderen Nachrichten und Objekte werden nicht an den Knoten weitergeleitet oder verarbeitet und es wird eine *FailureMessage* zurückgeschickt. Um zu gewährleisten, dass Nachrichten gesendet werden und nicht verloren gehen, bedient sich die Implementierung an dem Lazy Pirate Pattern [Hin13]. Jeder **LinkLayer** schickt bei Erhalt einer Nachricht eine entsprechende Nachricht zurück, sodass der Sender der Nachricht über den Erhalt informiert wird. Wenn Nachrichten an den Listeningsocket eines **LinkLayers** geschickt werden, wird anschließend auf eine Nachricht gewartet, um zu überprüfen, ob die Nachricht angekommen ist. Die Wartezeit entspricht dem gesetzten *POLL\_TIMEOUT* (3.1) innerhalb der `ModuleConfig`. Sollte keine Antwort erhalten worden sein, wird der Sendevorgang wiederholt. Die

Anzahl an Wiederholungen ist in *POLL\_TRIES* (3.1) gesetzt. Erst nach dem fehlgeschlagenen Empfangen einer Nachricht des letztmöglichen Versuches wird die Nachricht verworfen.

### 3.6.2 Ausfallmanagement

Im originalen Protokoll wurde nicht beachtet, dass der **LinkLayer** ausfallen oder gestoppt werden kann. Dadurch kann es passieren, dass Pings oder andere Nachrichten an nicht mehr vorhandene **LinkLayer** geschickt werden. Durch das oben beschriebene Pattern wird erkannt, dass ein **LinkLayer** nicht erreicht wird. Wenn es zu diesem Fall kommt, ist die InReferenz eines Relays, welche durch die Ping Nachrichten geprüft werden soll, eindeutig korrumpiert und sollte nicht mehr existieren. Dementsprechend wird, wenn ein **LinkLayer** nicht erreichbar ist und es sich um eine PingAction handelt, für jede Ping Aktion in der PingAction eine InRelayClosedAction an den Knoten gesendet, sodass der **RelayLayer** die falsche InReferenz korrigieren kann.

Analog wird eine OutRelayClosedAction an den Knoten gesendet, wenn bei Nachrichten, welche über Relays geschickt werden, ein **LinkLayer** nicht erreichbar ist.

Dieses Vorgehen kann mit der Flag *CONSIDER\_AS\_CLOSED* (3.1) eingeschaltet werden. Standardmäßig ist diese Option ausgeschaltet, sodass die Nachrichten nur verworfen und keine Anpassungen an den Relays vorgenommen werden. Beide Optionen dieser Einstellungen haben ihre Vor- und Nachteile. Sollte die Option eingeschaltet sein kann es passieren, dass **LinkLayer** als geschlossen gelten, obwohl diese eigentlich erreichbar sein sollten. Dadurch wird ermöglicht, dass Relay Verbindungen gelöscht werden, die kritische Informationen für den Zusammenhang des Systems beinhalten. Jedoch werden tatsächlich nicht erreichbare **LinkLayer** erkannt. Die entsprechenden Verbindungen werden gelöscht und falsche Daten innerhalb des **RelayLayers** können bereinigt werden. Sollte die Option ausgestellt werden, werden garantiert keine Verbindungen von Relays gelöscht. Jedoch kann das dazu führen, dass sehr viele Nachrichten an **LinkLayer** geschickt werden, welche nicht erreichbar sind. Da auf eine Antwort gewartet wird, kann es passieren, dass diese Nachrichten das System enorm verlangsamen und so die Buffer volllaufen. Das führt dazu, dass notwendige Verbindungsinformationen an bestimmten Knoten erst sehr spät verarbeitet werden.

### 3.6.3 Nachrichtenweiterleitung

Da der **LinkLayer** alle Nachrichten entweder an den Knoten oder an den **RelayLayer** sendet, wird anschließend kurz erläutert wann welche Nachricht an welche Instanz weitergeleitet wird und wie diese Weiterleitung geschieht. Sollte eine Nachricht über den Listeningsocket an den **LinkLayer** geschickt worden sein, handelt es sich dabei immer um eine TransmitMessage oder eine LayerMessage. Da diese Nachrichten für den **RelayLayer** bestimmt sind, müssen diese vom **RelayLayer** verarbeitet werden. Damit dieser die Nachricht bekommt, werden diese Nachrichten in den **RelayLayer** Buffer, welcher beim Erstellen eines **LinkLayer** Objekts mit *relay\_layer\_buffer* angegeben wird, eingefügt. Der **RelayLayer** kann dann selbst entscheiden, wann und wie er diese Nachricht bearbeitet. Der einzige Fall bei dem der **LinkLayer** eine Nachricht an den Knoten senden muss ist, wenn eine Aktion in den Buffer eines Sink-Relays eingefügt wird. Sollte der **LinkLayer** also eine Aktion innerhalb eines Buffers von einem Sink-Relay erhalten, wird diese in den Node Buffer eingefügt, welcher beim Erstellen des **LinkLayer** mit *node\_buffer* angegeben wird. Wichtig dabei ist, dass es sich immer um eine Action Nachricht handeln muss. Sollte eine TransmitMessage bei einem Sink-Relay ankommen, wird diese ebenfalls an den **RelayLayer** mit dem oben beschriebenen Verfahren gesendet.

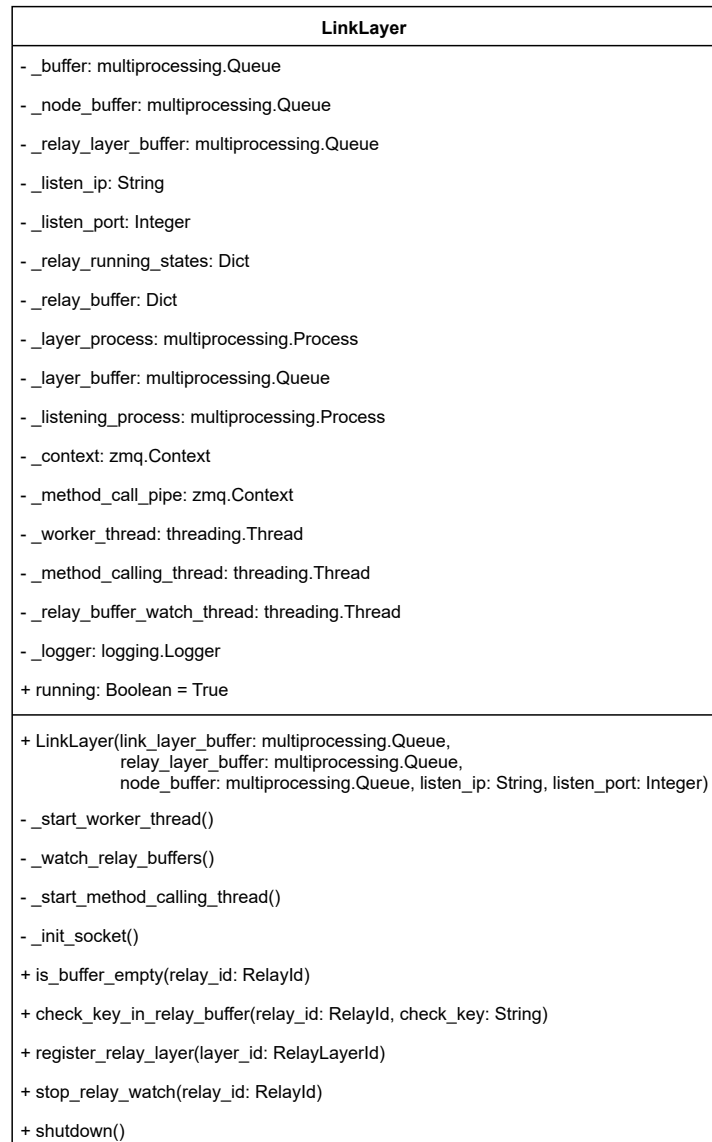


Abbildung 3.5: Klassendiagramm LinkLayer

### 3.6.4 Funktionen außerhalb der Klasse

Innerhalb des `LinkLayer` Moduls existieren vier Funktionen, welche außerhalb der Klasse existieren.

Die Funktion `start_link_layer` erstellt ein neues `LinkLayer` Objekt und wartet so lange, bis die `running` Flag des Objektes auf `False` steht. Diese Funktion wird benutzt, um den `LinkLayer` innerhalb eines Prozesses zu starten. Als Parameter werden bei der Funktion dieselben benötigt, wie die Parameter des Konstruktors der `LinkLayer` Klasse.

Die `start_listening(listen_ip, listen_port, relay_layer_buffer, running)` Funktion startet einen Socket des Typs `REP` und, solange der `running` Wert auf eins steht, wartet die Funktion auf Nachrichten. Die Adresse, auf welche der Socket hört, wird mit der `listen_ip` und dem `listen_port` angegeben. Wenn Nachrichten empfangen wurden, werden diese in die angegebene `RelayLayer` Queue `relay_layer_buffer` eingefügt. Zudem überprüft die Funktion, ob die empfangene Nachricht eine `LayerMessage` oder eine `TransmitMessage` ist und sendet eine `SuccessMessage` oder andernfalls eine `FailureMessage` zurück. Auch diese Funktion wird durch den `LinkLayer` inner-

halb eines Prozesses ausgeführt.

*start\_layer\_buffer\_watch(layer\_id, buffer, node\_buffer, running)* überwacht, solange der *running* Wert auf eins ist, die mit *buffer* angegebene Queue und leitet Nachrichten an andere **LinkLayer** weiter oder, wenn die Ziel **RelayLayerId** gleich der angegebenen *layer\_id* ist, fügt die Nachricht in die *relay\_layer\_buffer* Queue ein. Zudem wird, bevor die Nachricht an den **RelayLayer** geschickt wird, überprüft, ob die Nachricht eine **LayerMessage** ist.

Die letzte Funktion ist die *send\_message(context, layer\_id, message)* Funktion, welche die angegebene *message* an den Endpunkt sendet, welcher mit der *layer\_id* als **RelayLayerId** angegeben wurde. Gesendet wird diese Nachricht über einen Socket, welcher innerhalb des angegebenen *zmq.Context context* als *REQ* Socket erstellt wird. Innerhalb der Funktion wird mit dem in Abschnitt 3.6.1 beschriebenen Pattern überprüft, ob die Nachricht ankommt. Sollte die Nachricht erfolgreich zugestellt worden sein, wird *True* zurückgegeben und andernfalls *False*.

### 3.6.5 Buffer Designentscheidungen

Nach originalem Protokoll dient der **LinkLayer** dafür, die in dem **RelayLayer** befindlichen Buffer des **RelayLayers** und der Relays zu überwachen. In der Implementierung wurde sich nun dafür entschieden, dass der **LinkLayer** selbst die Buffer verwaltet. Der Grund dafür ist, dass der **LinkLayer** durch diese Entscheidung innerhalb eines eigenen Prozesses laufen kann und somit das Überwachen der Buffer und Senden von Nachrichten nicht die eigentlichen Prozesse des Knoten und des **RelayLayers** verlangsamt.

Daraus folgt jedoch, dass der **RelayLayer** keinen direkten Zugriff auf die Buffer der Relays besitzt. Somit müssen Informationen von den Buffern über Funktionen des **LinkLayers** abgefragt werden. Diese Methoden können dann von außerhalb über die *method\_call\_pipe* aufgerufen werden. Im Protokoll des **RelayLayer** sind zwei Abfragen über Informationen von Relay Buffern gefordert. Die erste Abfrage benötigt Informationen über den Füllstatus des Buffers eines Relays. Dafür existiert die *is\_buffer\_empty(relay\_id)* Methode innerhalb des **LinkLayers**, welche *True* zurückgibt, wenn der Buffer des angegebenen Relays leer ist, und andernfalls *False*. Die zweite benötigte Abfrage wird für die Verarbeitung von Probe Nachrichten benötigt und überprüft, ob in einem Buffer eines Relays Relay-Parameter existieren, welche einen bestimmten Key beinhalten. Für diese Abfrage gibt es die *check\_key\_in\_relay\_buffer(relay\_id, check\_key)*. In dieser Methode werden alle Nachrichten innerhalb des Buffers des angegebenen Relays überprüft, ob diese einen Relay-Parameter mit den angegebenen Key besitzen. Sollte so ein Relay-Parameter in einer Nachricht innerhalb des Buffers existieren, wird *True* zurückgegeben und andernfalls *False*.

Eine weitere Entscheidung, die getroffen werden musste, ist, dass die Überwachung des **RelayLayer** Buffers innerhalb eines Prozesses geschieht und die Überwachung der Relays innerhalb eines Threads. Dies hat zur Grundlage, dass erstens die Anzahl an Nachrichten, die in den **RelayLayer** Buffer eingefügt werden, wesentlich größer ist als die Anzahl der Nachrichten, welche über Relays weitergeleitet werden, sodass die geringe Anzahl der Relay Nachrichten die Ausführung des **LinkLayer** Prozesses nicht wesentlich verlangsamt. Zweitens können weiterhin Informationen über die Relay Buffer direkt über den **LinkLayer** Prozess abgefragt werden, ohne in einen weiteren Prozess eingreifen zu müssen.

### 3.7 RelayLayer

Innerhalb der `RelayLayer.py` Datei befindet sich die Klasse `RelayLayer`, welche die Aufgaben des Relay-Layers des originalen Protokolls erfüllt. Die Klasse ist für die Handhabung und Kontrolle der Relays des Systems beziehungsweise des zugrundeliegenden Knoten verantwortlich. Beim Erstellen des `RelayLayers` muss erstens eine `RelayLayerId` angegeben werden, welche den `RelayLayer` definiert, wobei die IP und der Port innerhalb der `RelayLayerId` den Endpunkt angibt, unter dem der `RelayLayer` erreichbar sein soll. Zweitens wird beim Erstellen eine `multiprocessing.Queue` als Parameter benötigt, in die alle Aktionen eingefügt werden, welche an den Knoten geschickt werden sollen. Während des Erstellens, werden drei Threads gestartet. Der erste Thread ist der Timeout Thread, welcher periodisch die Timeout Methode aufruft. In welchem Abstand die Timeout Funktion aufgerufen werden soll, wird, wie in Abschnitt 3.1 beschrieben, in der Modulkonfiguration definiert und in dem Attribut `timeout_period` gespeichert. Der zweite Thread ist ein Thread, der Nachrichten, welche in Buffer eingefügt werden sollen, an den `LinkLayer` schickt, damit dieser die Nachrichten in die entsprechenden Buffer einordnet. Im Prinzip werden alle Nachrichten, die in Buffer eingefügt werden, in die `buffer_put_actions` Queue eingefügt, welche dann von dem oben genannten Thread wieder herausgenommen wird. Der dritte Thread ist der Messagehandling Thread, welcher Nachrichten, die vom `RelayLayer` verarbeitet werden müssen und in den entsprechenden Buffer eingefügt wurden, aus der Queue entfernt und anschließend bearbeitet.

Weiterhin startet jeder `RelayLayer` einen Prozess, welcher den `LinkLayer` beinhaltet. Da sich der `LinkLayer` in einem anderen Prozess befindet als der `RelayLayer` selbst und der `RelayLayer` Funktionen im `LinkLayer` aufrufen muss, brauchen wir eine Kommunikationsmöglichkeit zwischen den beiden Prozessen.

Dafür verwendet der `LinkLayer` und der `RelayLayer` eine `multiprocessing.Pipe`. Diese wird, wie in Abschnitt 3.6 für den `LinkLayer` beschrieben, für den Aufruf von Methoden im `LinkLayer` verwendet. Damit nicht zwei Threads gleichzeitig auf die Pipe zugreifen, wird bei einem Methodenaufruf über die Pipe diese mithilfe eines Threading Locks, welcher in dem Attribut `calling_lock` gespeichert wird, gesichert.

Alle Relays, welche durch den `RelayLayer` verwaltet werden, werden innerhalb des Objekts in einem Dict gespeichert, wobei der Key des Eintrags die `RelayId` und der Wert das Relay Objekt ist.

Zu den eigentlichen Funktionen des Relay Layers kommen noch die Methoden, welche die Transmittaten der Relays überwachen. Diese Funktionalitäten sind für die Erkennung und Eindämmung von DoS Angriffe implementiert worden und werden in Abschnitt 4 näher erläutert.

#### 3.7.1 Grundlegende Änderungen

Als wesentliche Änderung gegenüber dem Pseudocode in den Referenzen [SS18] und [Set20] ist die Merge Funktion zu benennen. In der originalen Merge Beschreibung wird beim Zusammenfügen von Relays ein neues Relay erstellt, in das alle Informationen der anderen Relays eingefügt werden. Aus Implementierungsgründen wurde die Merge Funktion jedoch so umgesetzt, dass ein Relay (in diesem Fall immer das erste) aus der angegebenen Liste als Merge Relay ausgewählt wird, in das alle Informationen von den anderen Relays eingefügt werden. Semantisch sind die beiden Methoden gleich und es resultiert am Ende ein Relay mit den gleichen Informationen. Der wesentliche Vorteil der eben beschriebenen Methode ist, dass für ein Merge keine neue `RelayId` vergeben werden muss. Dadurch wird der interne Zähler `last_id`, welcher beim Erstellen einer neuen Relay ID hochgezählt wird, nicht erhöht und die Relay IDs bleiben niedrig.

Eine weitere Änderung ist die Auslagerung von Buffern in den `LinkLayer`. Die Gründe für die Änderungen wurden bereits in Abschnitt 3.6.5 erläutert.

Weiterhin wurden Methoden hinzugefügt, welche für die Validierungflag eines Relays benötigt werden. Darunter zählt die Methode *validate\_relay(relay\_id)*, welche ein Relay validiert und die Flag auf *True* setzt. Außerdem wurde die Methode *get\_validated\_relays* hinzugefügt, welche nur validierte Relays zurückgibt. Beim Implementieren ist zudem aufgefallen, dass der **RelayLayer** Relays löschen kann, ohne dass der Knoten eine Information darüber erhält. Somit führte eine anfängliche Version der Implementierung dazu, dass versucht wurde Nachrichten über bereits gelöschte Relays zu schicken. Um das zu beheben, wurde die Methode *check\_relay\_exists(relay\_id)* hinzugefügt. Die Methode gibt *True* zurück, wenn das angegebene Relay im **RelayLayer** existiert und *False* im entgegengesetzten Fall. Wie genau die Methode benutzt wird, wird später in Abschnitt 3.8.2 näher erläutert.

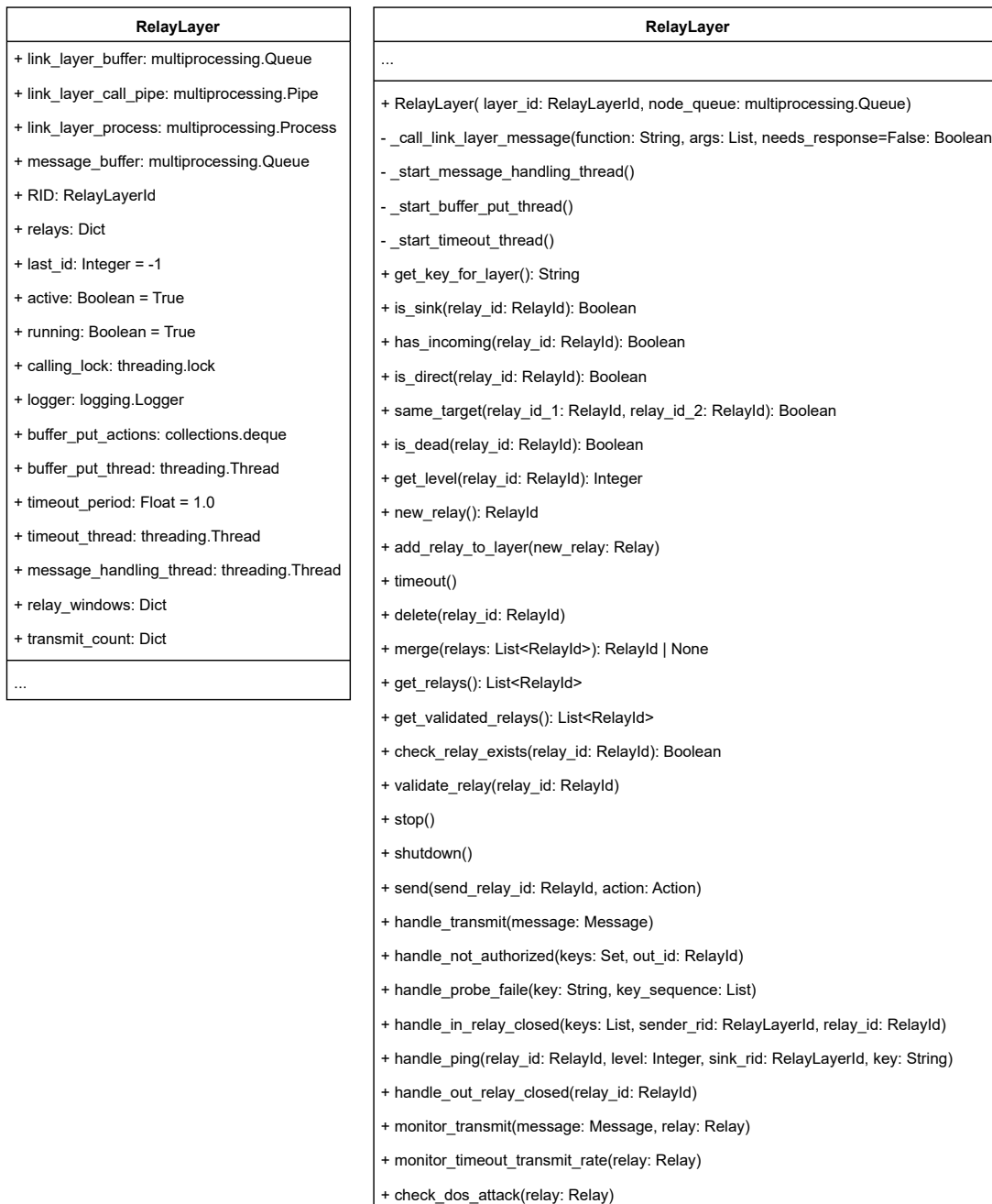


Abbildung 3.6: Klassendiagramm der RelayLayer Klasse

### 3.8 Node

Die Node Klasse, welche sich innerhalb der Node.py Datei befindet, ist die Hauptklasse, um selbststabilisierende Protokolle mit der Implementierung des Relay-Modells zu erstellen. Sie implementiert die Funktionalitäten, welche benötigt werden, um Aktionen zu verarbeiten und das System sicher verlassen zu können. Generell wird beim Erstellen eines Node Objekts eine Knoten ID benötigt, welche gebraucht wird, um eine Topologie bilden zu können, sowie eine IP und einen Port. Die IP und der Port werden benötigt, um die Adresse der Knoten beziehungsweise des `LinkLayers` zu definieren und erreichbar zu machen. Optional kann beim Konstruktor zusätzlich der Analysemodus eingeschaltet werden. Sollte der Analysemodus aktiviert sein, sendet der Knoten bei jedem Timeout einen Status seines Zustands an einen StateMonitor, wobei die Statusübermittlung überschrieben werden muss. Das Überschreiben der Funktionen wird nachfolgend in Abschnitt 3.8.1 beschrieben. Zudem erstellt jeder Knoten ein `RelayLayer` Objekt, der für diesen Knoten zuständig ist und mit dem der Knoten kommunizieren kann, um Nachrichten zu senden oder Relays zu erstellen.

Weiterhin werden beim Erstellen des Objekts zwei Threads erstellt. Der erste Thread überwacht den Nachrichten Buffer des Knoten, in den Aktionen eingefügt werden, wenn der `LinkLayer` welche empfängt. Diese Nachrichten werden nacheinander abgearbeitet. Der zweite Thread ist der Timeout Thread, welcher periodisch eine Timeout Aktion in den Node Buffer einfügt, so dass regelmäßig die Timeout Funktion des Knoten aufgerufen wird, wenn diese in dem Message Thread verarbeitet wird.

Die Timeout Funktion der Node Klasse entspricht der Implementierung des Pseudocodes aus den Listings 6.10 und 6.11 der Referenz [Set20].

#### 3.8.1 Benutzung des Knotens

Um selbststabilisierende Protokolle mithilfe der Node Klasse zu implementieren, muss eine Klasse erstellt werden, welche von dieser Klasse erbt und einige Funktionen überschreibt. Als Beispiel kann hierfür die Klasse `SortedListNode` aus der `SortedListNode.py` Datei betrachtet werden. Diese Klasse wurde für die Simulation einer Sortierten Liste erstellt. Die Analyse und deren Ergebnisse werden später in Abschnitt 5 erläutert.

Um eine Klasse für andere Protokolle zu erstellen, muss die Funktion *original\_timeout()* überschrieben werden. Diese Funktion wird am Ende des Timeouts der Node Klasse aufgerufen, wenn der Knoten das System nicht verlassen möchte. Diese Funktion soll die Funktionalität implementieren, welche das originale Protokoll für die Timeout Funktion vorsieht. Bei der Sortierten Liste ist dies beispielsweise die Korrektur der linken und rechten Nachbarn. Weiterhin kann jede Aktion als Funktion in dieser Klasse implementiert werden. Die *linearize* Methode der `SortedListNode` Klasse ist ein Beispiel hierfür. Sollte der Knoten eine Aktion *a* vom `LinkLayer` bekommen, wird *replace\_action(a)* aufgerufen, welche die Aktion *a* aufruft, wenn der Knoten das System nicht verlässt. Methoden, welche zusätzlich im Knoten implementiert werden, können mithilfe der Funktion *call\_method(relay, method, parameters)* aufgerufen werden. Dabei wird automatisch eine Aktion über das angegebene Relay gesendet, die die angegebene Methode mit den übergebenen Parametern in dem entsprechenden Knoten ausführt. Zudem braucht die Timeout Funktion der Node Klasse Zugriff auf die Relays, welche sich in den Variablen im originalen Protokoll befinden. Um dies zu gewährleisten, besitzt die Node Klasse die Liste *N*. Dort können entweder Relays hinzugefügt und entfernt, oder die Methoden *check\_in\_original\_variables(relay)*, *remove\_from\_original\_variables(relay)* und *get\_relays\_from\_original\_variables()* überschrieben werden.

Mit der *check\_in\_original\_variables(relay)* Methode wird im Node Timeout geprüft, ob ein bestimmtes Relay in einer Variable des originalen Protokolls vorhanden ist. Bei der Sortierten

Liste sind diese Variablen die linken und rechten Nachbarn. Der Node Timeout versucht zudem in bestimmten Fällen Relays von den originalen Protokollvariablen zu entfernen.

Dieses Verhalten kann mithilfe der *remove\_from\_original\_variables(relay)* definiert werden und sollte überschrieben werden.

Weiterhin muss die Node Klasse eine Liste von Relays aus den originalen Protokollvariablen erhalten können. Dafür existiert die *get\_relays\_from\_original\_variables()* Methode. Diese sollte eine Liste von allen Relays zurückgeben, welche sich in den Variablen des zugrundeliegenden Protokolls befinden. Alle drei Funktionen wurden in der SortedListNode Klasse überschrieben und können als Beispiel betrachtet werden. Eine wichtige Methode, welche überschrieben werden muss, um die Funktionalität des Protokolls zu gewährleisten, ist die *reversal\_of\_relay(relay)* Methode. Diese definiert eine Relay Umkehrung. Dabei muss der eigene Sink-Relay, welcher in *Node.in\_ref* gespeichert wird, als Parameter in einer Aktion des zugrundeliegenden Protokolls über das angegebene Relay gesendet werden. Dadurch wird eine Verbindung des Knotens, der die Aktion erhält, zum eigenen Knoten hergestellt. In dem Beispiel der Sortierten Liste, wird eine *linearize(in\_ref)* Aktion über das Relay gesendet. Dies wird durch den Aufruf von *call\_method(relay, 'linearize', [in\_ref])* ausgeführt.

Am Ende eines Timeout Durchlaufs der Node Klasse wird, wenn der Analysemodus aktiviert wurde, die Methode *send\_analyse\_state()* aufgerufen. Standardmäßig führt die Methode nichts aus. Diese kann aber beim Erstellen einer Unterklasse überschrieben werden, um eine Analyse zu verarbeiten. Im Beispiel der SortedListNode Klasse, wird ein SortedListState Objekt erstellt und mit der Node Funktion *send\_state\_to\_monitor(state)* an einen State Monitor geschickt. Die genauere Beschreibung des Status und dem StateMonitor geschieht in Abschnitt 5.1.

Möchte ein Knoten das System verlassen, genügt es die *stop()* Methode aufzurufen. Dabei wird die *leaving* Flag auf *True* gesetzt und das Protokoll führt das Verlassen des Systems aus. Möchte man hingegen den Knoten stoppen ohne Rücksicht auf irgendwelche Verbindungen, führt man die *shutdown()* Methode aus, die den Knoten stoppt und alle Layer herunterfährt.

### 3.8.2 Änderungen

Wie bereits in Abschnitt 3.7.1 beschrieben, trat das Problem auf, dass Relays gelöscht werden, ohne dass der Knoten darüber informiert wird. Mit der im **RelayLayer** hinzugefügten Methode *check\_relay\_exists* können wir abfragen, ob ein Relay innerhalb einer Variable des Knotens existiert oder bereits gelöscht wurde. Diese Abfrage wurde in der Timeout Methode des Knotens für alle Variablen, welche ein Relay beinhalten, durchgeführt. Sollte ein Relay nicht mehr existieren, wird dieses aus der entsprechenden Variable entfernt.

Weiterhin wurde die Timeout Methode in der Hinsicht angepasst, dass keine Iterationen mehr über alle Relays durchgeführt werden, sondern, wie in Abschnitt 3.4 beschrieben, nur über validierte Relays. Somit konnte verhindert werden, dass Relays gelöscht werden, bevor diese vom Knoten verarbeitet wurden.



Node
<ul style="list-style-type: none"> <li>+ buffer: multiprocessing.Queue</li> <li>+ relay_layer: RelayLayer</li> <li>+ running: Boolean = True</li> <li>+ timeout_thread: threading.Thread</li> <li>+ message_thread: threading.Thread</li> <li>+ logger: logging.Logger</li> <li>+ timeout_period: Integer</li> <li>+ leaving: Boolean = False</li> <li>+ in_ref: RelayId</li> <li>+ N: List</li> <li>+ D: List</li> <li>+ a_out: RelayId</li> <li>+ a_in: RelayId</li> <li>+ analyse_mode: Boolean</li> <li>- _context: zmq.Context()</li> </ul>
<ul style="list-style-type: none"> <li>+ Node(node_id: Integer, ip: Integer, port: Integer, analyse_mode: Boolean=False)</li> <li>- _start_timeout_thread()</li> <li>- _start_message_handling()</li> <li>+ timeout()</li> <li>+ original_timeout()</li> <li>+ reversal_of_relay(relay_id: RelayId)</li> <li>+ check_in_original_variables(relay_id: RelayId)</li> <li>+ remove_from_original_variables(relay_id: RelayId)</li> <li>+ get_relays_from_original_variables()</li> <li>+ send_analyse_state()</li> <li>+ replace_action(action: Action)</li> <li>+ ask_to_reverse(out: RelayId)</li> <li>+ ask_to_reverse_anchor(out: RelayId, receiving_relay: RelayId)</li> <li>+ notify_anchor()</li> <li>+ reverse(out: RelayId)</li> <li>+ call_method(out: RelayId, method: String, parameters: List)</li> <li>+ start()</li> <li>+ send_state_to_monitor(state)</li> <li>+ stop()</li> <li>+ shutdown()</li> </ul>

Abbildung 3.7: Klassendiagramm der Node Klasse

### 3.9 Keys, Logging und Überblick

Weiterhin bleibt zu klären, wie die Keys, welche zur Authentifizierung von Verbindungen genutzt werden, gebildet werden. Dafür existieren innerhalb der `KeyGeneration.py` Datei einige Funktionen, die die Handhabung von Keys ermöglichen. Das originale Protokoll erfordert, dass ein Teil des Keys darauf geprüft werden kann, ob der Key von einem bestimmten `RelayLayer` generiert wurde. Um diese Voraussetzung zu erfüllen, besteht der Key aus zwei Teilen. Der erste Teil repräsentiert den Prefix, welcher die RID eines `RelayLayers` bei der Generierung beinhaltet. Dieser wird gebildet, indem ein `sha1` Hash mit der String Repräsentation der RID erstellt wird. Der zweite Teil des Keys besteht aus einer zufällig generierten ID. Diese ID wird mithilfe der vierten Version der UUID Spezifikation erstellt [LMS05]. Dafür wird die durch Python gegebene UUID Klasse mit der `uuid4` Funktion verwendet [Fou]. Getrennt werden die beiden Teile durch ein Rauten Symbol.

Um einen Key zu generieren, genügt es die Funktion `generate_key(layer_id)` aufzurufen. Die angegebene `layer_id` entspricht dabei einem `RelayLayerId` Objekt. Separat dazu kann mithilfe der `generate_prefix(layer_id)` Funktion lediglich der Prefix eines Keys generiert werden. Auch hier entspricht die angegebene `layer_id` einem `RelayLayerId` Objekt. Um zu überprüfen, ob ein bestimmter Key zu einer bestimmten RID passt, kann die `check_key_origin(key, layer_id)` Funktion verwendet werden. Diese überprüft, ob der Prefix des angegebenen Keys zu der angegebenen RID passt. Sollte der Prefix zu der RID passen wird `True` zurückgegeben und andernfalls `False`.

Generell gestaltet sich das Debuggen des Frameworks aufgrund der ausgelagerten Prozesse als schwierig. Um das Problem zu lösen, besitzt jede Klasse einen Logger. Die Logdateien der Klassen werden innerhalb des „logs“ Ordners geschrieben. Standardmäßig werden nur Warnungen oder schwerwiegendere Logging Einträge in die Logs geschrieben. Dieses Level kann aber innerhalb der Modulkonfiguration, wie in Abschnitt 3.1 vorgestellt, geändert werden, damit auch Debug Informationen ausgegeben werden. Um für weitere Klassen einen Logger einzubinden, wurde innerhalb der `RelayLogging.py` Datei die Funktion `get_logger(log_level, file_name)` hinzugefügt. Diese erlaubt es, ein Logger Objekt zu erhalten, welches die Logs im selben Format und denselben Ordner schreibt. Dabei ist jedoch zu beachten, dass der angegebene Dateiname eindeutig für das Objekt sein sollte, damit es zu keiner Kollision von Logger Objekten kommt. Zudem sollte der angegebene Loglevel, sofern nicht anders gewollt, dem Loglevel entsprechen, welches in der Konfiguration gesetzt wurde.

Nachfolgend wird der generelle Aufbau des Frameworks und die Kommunikation zwischen den Prozessen in der Abbildung 3.8 zusammenfassend dargestellt. Dabei wurde in der Abbildung auf Threads verzichtet, die keinen Einfluss auf die Kommunikation zwischen den Schichten besitzen. Zudem wurden die Klassen, Methoden und Objekte vereinfacht dargestellt. Die Abbildung zeigt die Prozesse im Zusammenspiel mit der Kommunikation eines Knotens.

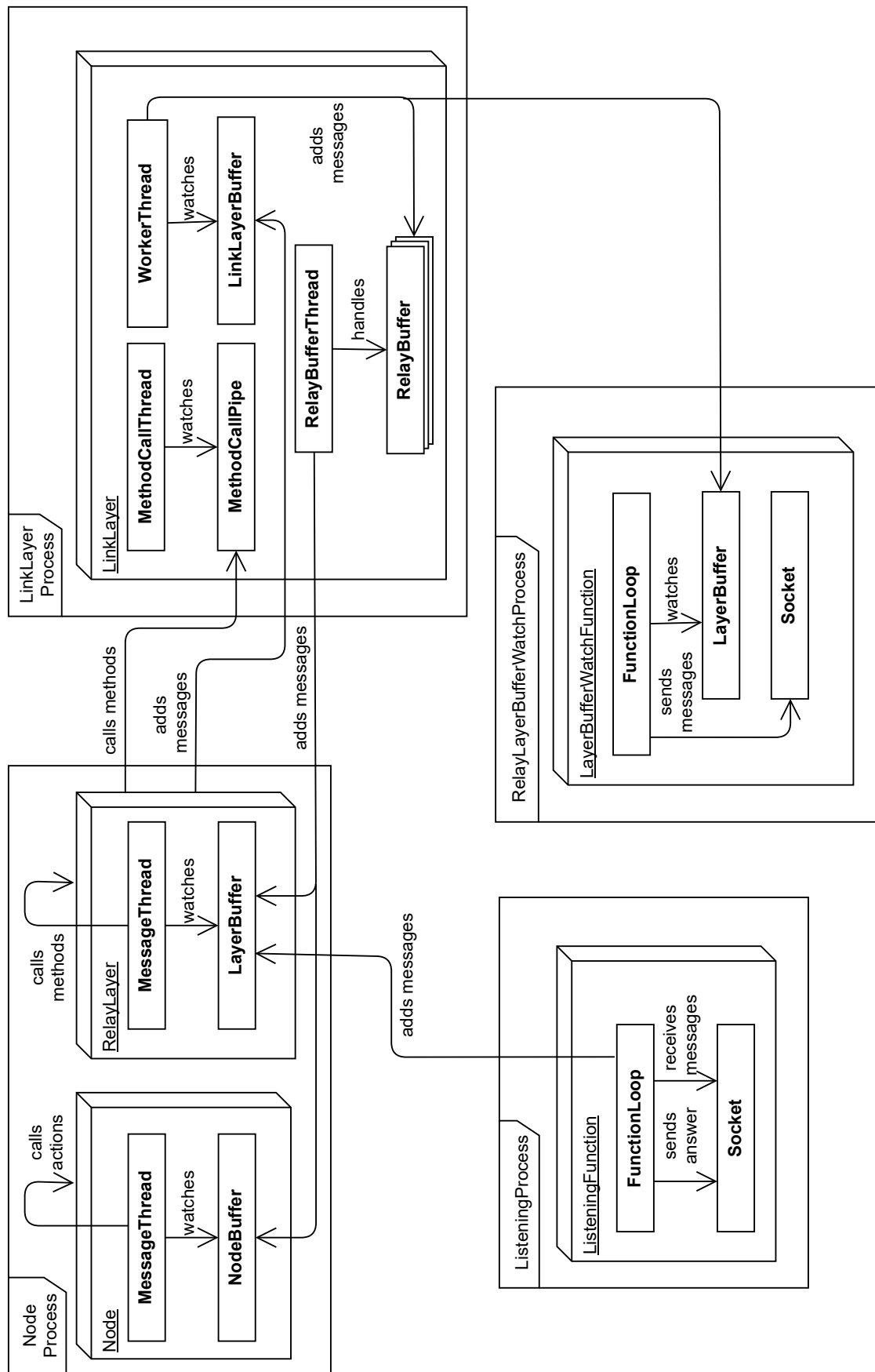


Abbildung 3.8: Übersicht der Kommunikation des Frameworks



## Denial of Service Erweiterungen

Ein weiterer Bestandteil dieser Arbeit ist es, das Framework so zu erweitern, dass DDoS Angriffe erkannt und gegebenenfalls verhindert werden können. Das Modell gibt bereits voraus, dass ein Knoten beziehungsweise der zugrundeliegende Relay-Layer entscheiden kann, welche Knoten eingehende Verbindungen besitzen dürfen. Es bleibt lediglich die Erkennung von diesen Angriffen zu entwickeln und zu implementieren. Anschließend werden zuerst die Algorithmen vorgestellt und deren Ablauf skizziert. Nachfolgend wird beschrieben, wie die Algorithmen in der RelayLayer Klasse eingebunden und benutzt werden können.

### 4.1 Erkennungsalgorithmen

Generell erfassen die Algorithmen zur Erkennung der Angriffe grundlegenden Änderungen in den Übertragungsraten. Jegliche Übertragungen über Relays geschehen über Transmittnachrichten, welche zuerst durch den Relay-Layer verarbeitet werden. Wenn wir nun die Anzahl der Übertragungen über ein bestimmtes Relay zählen, können wir bei jedem Timeout Aufruf des Relay-Layers eine Übertragungsrate des bestimmten Relays ermitteln. Die Rate entspricht dann der Anzahl der Übertragungen des Relays pro Timeout. Bevor wir jedoch Änderungen in den Raten erkennen können, müssen wir diese effizient speichern. Um die Menge an Daten gering zu halten, wird für die Speicherung eine Queue mit einer festen Länge verwendet. Diese Queue funktioniert dann wie ein Schiebefenster auf den gesamten Raten. Sollte eine neue Rate in die Queue eingefügt werden, wird diese hinten angehängt. Wenn die Länge durch das Einfügen des neuen Werts überschritten werden sollte, wird am anderen Ende der erste Wert wieder herausgelöscht.

Nachfolgend werden die Algorithmen vorgestellt und deren Korrektheit skizziert. Der erste Algorithmus ist die *MonitorTransmit* Funktion, welche die Übertragungen der Relays zählt. Der Pseudocode der Funktion ist in dem Algorithmus 1 zu sehen. Die Funktion speichert die Übertragungen, welche keine Standardnachrichten sind, für ein angegebenes Relay  $r$  innerhalb eines Zählers.

*Beweisskizze.* Angenommen die *MonitorTransmit* Funktion wird bei jeder Übertragung einer Nachricht ausgeführt und das Relay und die Nachricht werden als Parameter übergeben. Sollte die Nachricht beim Aufruf der *MonitorTransmit* Funktion eine Standardnachricht sein, so wird nichts durchgeführt und keine Übertragung wird registriert. Wenn jedoch eine Nachricht angegeben worden ist, welche keiner Standardnachricht entspricht, wird zuerst geprüft, ob bereits ein Zähler für das angegebene Relay existiert. Sollte kein Zähler existieren, wird ein Zähler für

**Algorithm 1** MonitorTransmit

---

```

1: function MONITORTRANSMIT(Message  $m$ , Relay  $r$ )
2:   if  $m$  ist keine Standardaktion then
3:     if  $\nexists$  kein TransmitCount für  $r$  then
4:       TransmitCount[ $r$ ]  $\leftarrow$  1
5:     else
6:       TransmitCount[ $r$ ] + 1
7:     end if
8:   end if
9: end function

```

---

$r$  erstellt und mit eins initialisiert. Bei jedem weiteren Aufruf der Funktion für das Relay  $r$  und einer nicht Standardnachricht, wird der vorher erstellte Zähler um eins hochgezählt. Somit steht in dem Zähler für  $r$  immer die Anzahl an Übertragungen von nicht Standardnachrichten.

Der nächste Algorithmus ist die *MonitorTransmitRate* Funktion, welche im Algorithmus 2 dargestellt wird. Diese Funktion ermittelt die Übertragungsrate eines Relays und fügt diese in ein entsprechendes Schiebefenster ein. Somit speichert die Funktion die  $n$  letzten Übertragungsraten eines angegebenen Relay  $r$  bei einer Fenstergröße von  $n$ .

**Algorithm 2** MonitorTransmitRate

---

```

1: function MONITORTRANSMITRATE(Relay  $r$ )
2:   if  $\nexists$  Fenster für  $r$  then
3:     Erstelle ein neues Schiebefenster für  $r$ 
4:   end if
5:    $winR \leftarrow$  Fenster von  $r$ 
6:    $winR \leftarrow winR + TransmitCount$ 
7:   if Länge von  $winR$  ist größer als die maximale Fenstergröße  $n$  then
8:     Entferne erstes Element von  $winR$ 
9:   end if
10:  TransmitCount[ $r$ ]  $\leftarrow$  0
11: end function

```

---

*Beweisskizze.* Angenommen bei jeder Übertragung über ein Relay  $r$  wird der interne Zähler *TransmitCount* mit *MonitorTransmit* hochgezählt und es existiert noch kein Schiebefenster für  $r$ . Dann wird beim ersten Aufruf ein Schiebefenster mit maximaler Länge  $n$  für  $r$  erstellt und die erste Anzahl an Übertragungen aus dem *TransmitCount* in das Schiebefenster eingefügt. Somit steht an der ersten Stelle die Übertragungen als Rate mit Übertragungen pro Funktionsaufruf. Wiederum werden die Übertragungen in dem *TransmitCount* gezählt. Bei jedem weiteren Aufruf der Funktion *MonitorTransmitRate* wird in das vorhandene Schiebefenster der aktuelle Übertragungszähler eingefügt. Somit sind nach  $n$  Aufrufen der Funktion  $n$  Übertragungsraten in das Schiebefenster eingefügt worden. Sollte die Funktion  $n + 1$  Mal aufgerufen werden, wird das erste Element aus dem Schiebefenster wieder entfernt und es bleiben genau  $n$  Werte im Fenster bestehen. Analog dazu wird bei jedem weiteren Aufruf ein neuer Wert hinten angehängt und das erste Element entfernt. Somit bleiben immer die letzten  $n$  Übertragungsraten innerhalb des Schiebefensters gespeichert.

Für die Erkennung des Angriffs wurde sich für einen Threshold-Ansatz entschieden, da bei der Recherche andere Möglichkeiten wie zum Beispiel ein Volumen-Ansatz, bei dem ein Volu-

men zuvor festgelegt wird und ein Angriff erkannt wird, wenn das Volumen ausgereizt wurde, nicht passend für das Relay-Modell erschienen. Grund dafür ist, dass ein Volumen bei langer Ausführungszeit immer ausgereizt wird. Somit könnten falsch-positive Erkennungen auftreten, wenn das System länger läuft.

Wenn wir nun überprüfen wollen, ob die gespeicherten Übertragungsraten eine signifikante Änderung beinhalten, können wir dies mithilfe der *CheckDoSAttack* Funktion durchführen. Diese Funktion teilt das Schiebefenster eines Relays in zwei Hälften und vergleicht die beiden Hälften. Sollte ein Strukturbruch vorliegen, wodurch die Raten signifikant ansteigen, wird *True* zurückgegeben und andernfalls *False*. Die zugrundeliegende Idee und der Algorithmus wurde von Kifer et al. 2004 [KBG04] entwickelt und von Bengs in der Vorlesung Data Mining 2020 [Ben20] vereinfacht wiedergegeben. In dieser Arbeit wurde der Algorithmus dementsprechend angepasst, dass Konzeptänderungen innerhalb eines Datenstroms nur in eine Richtung erkannt werden, denn uns interessiert nur eine signifikante Änderung in steigenden Übertragungsraten. Sollten die Übertragungsraten abrupt abfallen, ist dies kein Indiz für einen DoS Angriff und ist somit nicht von Relevanz. Der Pseudocode der *CheckDoSAttack* Funktion ist in dem Algorithmus 3 dargestellt. Die Funktion erkennt DoS Angriffe bei plötzlich auftretenden gleichmäßig hohen Übertragungsraten über ein bestimmtes Relay.

---

**Algorithm 3** CheckDoSAttack based on [KBG04, Ben20]

---

**Output:** DoS Detected

```

1: function CHECKDOSATTACK(Relay r)
2:    $winR \leftarrow$  Fenster von r
3:    $W_1 \leftarrow$  Erste Hälfte von  $winR$ 
4:    $W_2 \leftarrow$  Zweite Hälfte von  $winR$ 
5:    $mean_1 \leftarrow$  Arithmetisches Mittel von  $W_1$ 
6:    $mean_2 \leftarrow$  Arithmetisches Mittel von  $W_2$ 
7:    $var_1 \leftarrow$  Varianz von  $W_1$ 
8:    $var_2 \leftarrow$  Varianz von  $W_2$ 
9:    $changeRate \leftarrow \frac{mean_1 - mean_2}{\sqrt{\frac{2}{n} * (var_1 + var_2)}}$ 
10:  if  $changeRate < \alpha$  then
11:    return True
12:  else
13:    return False
14:  end if
15: end function

```

---

*Beweisskizze.* Angenommen es existiert ein Fenster für das Relay  $r$ , in dem die letzten  $n$  Übertragungsraten gespeichert sind. Weiterhin setzen wir  $\alpha$  auf einen negativen Wert der entsprechend gesetzt wird, damit Änderungen erkannt werden. Nehmen wir an, dass zum Zeitpunkt des Funktionsaufrufs kein Angriff durchgeführt wird, folgt daraus, dass die Übertragungsraten über das Relay  $r$  gleichmäßig verteilt sind und keine signifikanten plötzlichen Steigungen der Übertragungsraten existieren. Da nun die letzten  $n$  Übertragungsraten innerhalb des Fensters gespeichert sind und diese Übertragungsraten gleichmäßig sind, sollten sich die arithmetischen Mittel der beiden Hälften des Fensters nicht wesentlich voneinander unterscheiden. Daraus folgt, dass  $mean_1 - mean_2$  gegen Null geht und die  $changeRate$  somit auch gegen Null geht. Somit bleibt der berechnete Wert über dem gesetzten Threshold  $\alpha$  und es wird kein Angriff erkannt. Sollte nun ein Angriff durchgeführt werden, folgt daraus, dass über einen längeren Zeitraum sehr hohe Übertragungsraten am Relay  $r$  auftreten. Angenommen die hohen Übertragungsraten bleiben zudem gleichmäßig hoch. Wenn wir nun  $\frac{n}{2}$  Runden warten, stehen in der ersten Hälfte

des Schiebefensters Übertragungsraten, welche nicht aus einem Angriff kommen und dementsprechend gleichmäßig niedrig sind, und in der zweiten Hälfte die hohen Übertragungsraten aus dem Angriff. Somit ist das arithmetische Mittel der ersten Hälfte signifikant kleiner als das der zweiten Hälfte. Dadurch wird  $mean_1 - mean_2$  kleiner als 0 und somit die *changeRate* ebenfalls wesentlich kleiner als 0. Somit ist die errechnete Rate kleiner als  $\alpha$  und es wird korrekterweise ein Angriff erkannt. Sollte nun zufällig die Übertragungsrate für eine Runde signifikant gestiegen sein, ist die Varianz der zweiten Hälfte auch gestiegen, was dazu führt, dass der Nenner größer wird und dies die errechnete *changeRate* klein hält. Somit werden kurze hohe Übertragungsraten nicht als Angriff erkannt.

## 4.2 Einbindung in die RelayLayer Klasse

Um nun die Algorithmen in die **RelayLayer** Klasse einzubinden, damit dieser DoS Angriffe effizient erkennen kann, müssen die Standardmethoden des **RelayLayers** angepasst werden. Generell kann die DoS Überprüfung ausgeschaltet werden. Dafür existiert, wie in Abschnitt 3.1 bereits beschrieben, die Variable *DOS\_DETECTION\_ACTIVATED*. Bevor die Übertragungsraten analysiert werden können, müssen diese gespeichert werden. Dafür benötigen wir für jedes Relay ein Schiebefenster. Um das zu gewährleisten, speichert der **RelayLayer** innerhalb des Attributs *relay\_windows* ein Dictionary (Abb. 3.6). Dieses Attribut speichert für jedes Relay das entsprechende Schiebefenster als *collection.deque* Objekt. Dabei ist der Key des entsprechenden Schiebefensters die RelayId des zugehörigen Relays. Weiterhin benötigen wir, wie oben beschrieben, einen *TransmitCount*, welcher die Übertragungen über ein bestimmtes Relay speichert. Dafür besitzt die **RelayLayer** Klasse das Attribut *transmit\_count*, welches wiederum als Dictionary gespeichert wird. In diesem Dictionary werden für jedes Relay vom **RelayLayer** die Übertragungen hochgezählt. Dabei ist der Key die RelayId und der Wert die Übertragungen als Ganzzahl. Nun besitzt der **RelayLayer** alle Voraussetzungen für die Einbindung der oben beschriebenen Algorithmen.

Zuerst müssen wir die Übertragungen innerhalb des *transmit\_count* Attributs protokollieren. Dafür wird die Transmit Methode des originalen Protokolls, wie in Algorithmus 4 gezeigt, leicht verändert. Dadurch werden die Übertragungen für jedes Relay gespeichert.

---

### Algorithm 4 Transmit Funktion des RelayLayers

---

```

1: function TRANSMIT(Message m)
2:   if P besitzt ein Relay r mit r.alive = True und m hat einen validen Header für r then
3:     MonitorTransmit(m,r)
4:     # Transmit vom originalen Protokoll
5:   end if
6: end function

```

---

Mit den gespeicherten Übertragungen können wir diese jetzt periodisch analysieren. Dafür wird die Timeout Funktion des **RelayLayers** entsprechend angepasst. Zuerst werden aus den gezählten Übertragungen Übertragungsraten gebildet, indem die *MonitorTransmitRate* Funktion für das bestimmte Relay aufgerufen wird. Wenn wir nun die Erkennung von DoS Angriffen in der Konfiguration eingeschaltet haben, können wir die Übertragungsraten analysieren, indem wir die Funktion *CheckDoSAttack* aufrufen. Das Standardverhalten beim Erkennen eines DoS Angriffs ist das Löschen des Relays. Der Ablauf der neuen Timeout Funktion ist in Algorithmus 4 zu sehen.

Die in Abschnitt 4.1 vorgestellten Funktionen werden innerhalb des **RelayLayers** durch die Methoden *monitor\_transmit*, *monitor\_timeout\_transmit\_rate* und *check\_dos\_attack* im-



**Algorithm 5** Timeout des RelayLayers

---

```

1: function TIMEOUT
2:   for all relays  $r$ , die  $P$  besitzt do
3:     MonitorTransmitRate( $r$ )
4:     if DoS Erkennung aktiviert then
5:        $d \leftarrow$  CheckDoSAttack( $r$ )
6:       if  $d$  ist True then
7:         delete( $r$ )
8:       end if
9:     end if
10:    # Timeout vom originalen Protokoll
11:  end for
12: end function

```

---

plementiert.

Die Berechnung der *changeRate* innerhalb der *CheckDoSAttack* Methode wird jedoch nicht innerhalb des **RelayLayer** selbst berechnet, sondern wurde in die *ConceptChange.py* Datei ausgeleitet. In dieser Datei befindet sich die *calculate\_windows(window\_a, window\_b)* Funktion, welche die *changeRate* von zwei gleichgroßen Listen berechnet. Als Parameter werden die beiden Hälften des Fensters angegeben. Weiterhin möchten wir Übertragungen von Standardaktionen, welche für den **RelayLayer** reserviert werden, nicht mit in die Übertragungsraten einbeziehen. Dabei handelt es sich um Aktionen wie zum Beispiel die Probe und ProbeFail Nachrichten. Alle Aktionen, die dabei ignoriert werden sollen, können, wie in Abschnitt 3.1 vorgestellt, in der Konfigurationsvariable *NO\_MONITOR\_ACTIONS* angegeben werden. Dabei werden alle Nachrichten, deren Aktionstyp sich innerhalb dieser Liste befindet, in der *monitor\_transmit* Methode ignoriert. Daraus folgt, dass nur Übertragungen protokolliert werden, welche Aktionen des Knotens sind. Der Grund für das Ignorieren der Standardnachrichten liegt darin, dass Standardnachrichten nur vom **RelayLayer** verwendet werden sollten. Die Angriffe gehen von einem kontrollierten Knoten aus und ein Angreifer sollte keinen Zugriff auf den zugrundeliegenden **RelayLayer** besitzen. Dadurch können nur Nachrichten geschickt werden, welche durch den kompromittierten Knoten gesendet werden können. Die Standardnachrichten können somit nicht vom Angreifer verwendet werden und sind dadurch irrelevant für die Erkennung der Angriffe.



## Finite Departure Analyse

Um nun die erfolgreiche Funktion des Frameworks zu zeigen, führen wir mithilfe der Library eine Analyse des  $\mathcal{FDP}$  durch. Dafür werden nachfolgend zuerst die Implementierung der Simulation und wesentliche Methoden vorgestellt. Abschließend folgen die Ergebnisse der Analyse mit anschließender Diskussion.

### 5.1 Implementierung und Methodik der Analyse

Generell benötigt das  $\mathcal{FDP}$  ein zugrundeliegendes selbststabilisierendes Protokoll. Für die Analyse dieser Arbeit wurde sich dazu entschlossen, das BuildList Protokoll [FSS20] einer Sortierten Liste zu benutzen. Um das Protokoll auf das Framework anzuwenden, wurde die Klasse SortedListNode erstellt, welche, wie in Abschnitt 3.8.1 beschrieben, von der Node Klasse erbt und die wichtigen Funktionen überschreibt. Die zwei wesentlichen Methoden der Klasse sind die Timeout Methode, welche innerhalb der *original\_timeout* Methode implementiert wurde, und die Linearize Methode. Weiterhin werden die linken und rechten Nachbarn des Knoten innerhalb der *left* und *right* Attribute als RelayId gespeichert. Wichtig zu erwähnen ist, dass Relay-Referenzen, welche als Parameter in der Linearize Methode vorkommen, gemerged werden, wenn diese Referenz die gleiche Ziel NodeId besitzt wie eine Relay-Referenz in einer der Variablen des Knoten. Dieses Verhalten ist so nicht im originalen BuildList Protokoll gegeben und musste für die Benutzung mit dem Framework angepasst werden.

Um die Simulation effizient zu gestalten, wird jeder Knoten innerhalb eines eigenen Prozesses gestartet. Da wir dadurch aber keinen Zugriff mehr auf die Attribute eines Knoten besitzen, brauchen wir eine Möglichkeit das gesamte System zu überwachen, um zu entscheiden, wann das System in einen validen Zustand gelangt. Für das Vorhaben wurde die StateMonitor Klasse innerhalb der StateMonitor.py Datei entworfen. Diese Klasse ist einzig dafür verantwortlich alle Analysestatus von Knoten zu empfangen, zu speichern und periodisch zu prüfen, ob das System stabil ist. Weiterhin gibt der StateMonitor jedem Knoten Auskunft über die Validität des Systems. Der genaue Ablauf des StateMonitors wird später näher erläutert. Damit jeder Knoten einen Status sendet, kann der Analysemodus eines Knotens mit dem *analyse\_mode* Attribut eingeschaltet werden. Sollte dieser eingeschaltet sein, wird am Ende jedes Timeouts der Methode *send\_analyse\_state* aufgerufen. Um das System analysieren zu können, brauchen wir nicht alle Informationen eines Knoten, sondern nur die wesentlichen Informationen über Verbindungen und Knotenstatus. Dafür wurden die Wrapper-Klassen *SortedListNodeState* und *StateRelay* innerhalb der SortedListNode.py Datei erstellt. Die Klasse *SortedListNodeState* beinhaltet, wie

in Abbildung 5.1 zu sehen, die *NodeId* des Knoten, die Verbindung zum linken und rechten Nachbarn, alle Relays des Knoten, den *running* Status des Knoten und den Status, ob der Knoten das System verlassen möchte. Alle Verbindungen, welche als Relays in der *SortedListNodeState* Klasse gespeichert werden, sind ein Objekt der *StateRelay* Klasse. Diese beinhaltet alle wichtigen Informationen, um zu entscheiden, zu welchem Knoten eine Verbindung besteht. In dieser Klasse werden die *RelayId*, der *alive* Status, die *SinkRID* als *RelayLayerId* und die Information, ob das Relay ein direktes Relay ist oder nicht, gespeichert. Beim Initialisieren eines *StateRelay* Objekts wird ein *Relay* Objekt angegeben, aus dem automatisch die entsprechenden Informationen herausgenommen werden.

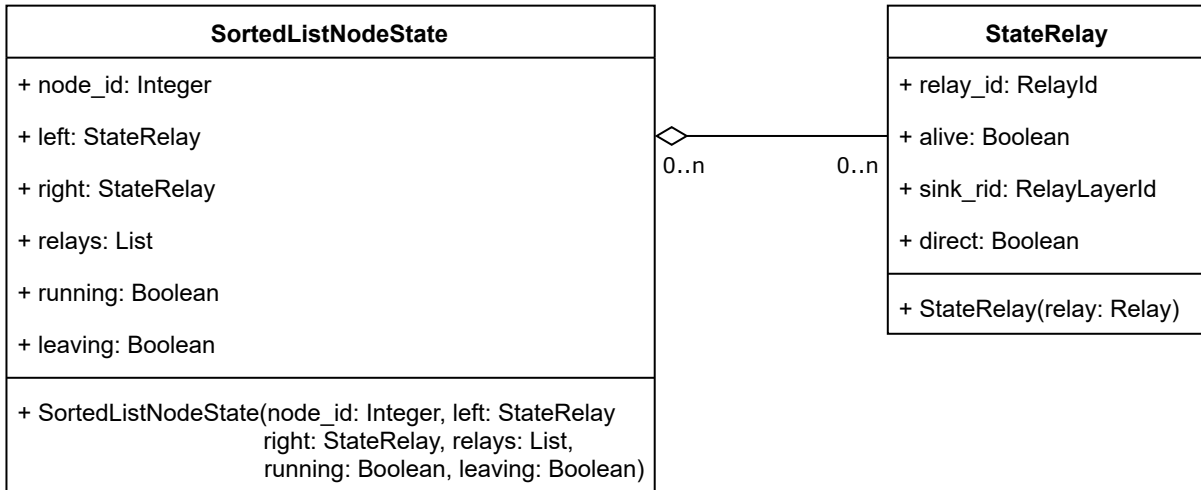


Abbildung 5.1: Analyse Klassen innerhalb der SortedListNode.py Datei

Mit diesen Klassen können wir nun die wesentlichen Informationen an den StateMonitor schicken. Dafür wurde in der SortedListNode Klasse die Methode *send\_analyse\_state* überschrieben. In dieser Funktion erstellen wir zuerst für den linken und rechten Nachbarn ein *StateRelay*. Weiterhin wird eine Liste von *StateRelays* erstellt, in der alle validierten Relays des zugrundeliegenden *RelayLayer* eingefügt werden. Aus den *StateRelays* und den Knoten Informationen wird dann ein *SortedListNodeState* Objekt zum Versenden erstellt. Zum Schluss wird dieser Status über die Methode *send\_state\_to\_monitor* an den StateMonitor gesendet und das Ergebnis des Systemstatus zurückgegeben. Das Versenden innerhalb der Methode geschieht über einen zeromq Socket des Typs *REQ*, welcher sich mit dem Endpunkt des StateMonitors verbindet. Die Adresse des StateMonitors wird, wie in Abschnitt 3.1 vorgestellt, in der Konfiguration angegeben. Die Methode gibt, nach dem Senden des Status, die Antwort des StateMonitors zurück, welcher normalerweise den Status des Systems repräsentiert.

Sollte der StateMonitor einem Knoten einen validen Systemzustand signalisieren, wird, mit dem aktuellen Stand der Node Implementierung die *shutdown* Methode aufgerufen und der Knoten wird beendet.

Nun bleibt noch zu klären, wie der StateMonitor den Systemzustand analysiert und validiert. Generell besitzt der StateMonitor, wie in Abbildung 5.2 zu sehen, zwei Threads. Der erste Thread ist der *listening\_thread*, welcher ein zmq Socket des *REP* Typs öffnet und auf diesem Nachrichten empfängt. Sollte eine Nachricht als *SortedListNodeState* Objekt auf diesem Socket ankommen, wird die Methode *\_handle\_state* aufgerufen, in der die Informationen aus dem State gespeichert werden. Sollte der Systemstatus *valid* auf *True* stehen, wird innerhalb des Threads gewartet bis jeder Knoten über den validen Systemzustand informiert wurde. Wenn es zu einer Phase ohne Nachrichteneingang kommt und alle Knoten informiert worden sind, wird die *stop* Methode aufgerufen und der StateMonitor gestoppt. Innerhalb der *\_handle\_state* Methode

werden zuerst der empfangene Status des Knoten gespeichert und anschließend die Timeouts mitgezählt. Bei jedem Empfangen eines Status wird der Timeoutzähler um eins hochgezählt, wodurch protokolliert werden kann, wie viele Timeouts jeder Knoten benötigt bis das System stabil ist. Der zweite Thread ist der *analyse\_thread*, welcher periodisch das System auf Validität prüft. Dabei wird das System erst geprüft, wenn der StateMonitor von allen Knoten im System einen Status erhalten hat. Dafür existiert das *node\_count* Attribut, welches definiert wie viele Knoten sich im System befinden und beim Initialisieren des Objekts angegeben wird. Um das System validieren zu können, wurden innerhalb der *Validation.py* Datei einige Funktionen erstellt, die prüfen, ob ein System aus angegebenen *SortedListNodeState* Objekten valide ist. Sollte innerhalb des Threads das System valide sein, wird das interne Attribut *valid* auf *True* gesetzt. Nachdem das System als valide angesehen wird, wird dieser Thread beendet und zum Ende die Methode *\_write\_result\_message* aufgerufen. Diese Methode schreibt alle Informationen über Timeouts und Knoten in entsprechende Dateien, damit diese später analysiert werden können. Der Ordner, in dem diese Datei geschrieben wird, kann, wie in Abschnitt 3.1 vorgestellt, in der Konfiguration angegeben werden. Der Name der Datei besteht immer aus der Anzahl an Knoten, einem Unterstrich und dem aktuellen Timestamp.

StateMonitor
- <i>_running</i> : Boolean = True - <i>_context</i> : zmq.Context + <i>node_count</i> : Integer + <i>node_timeout_count</i> : Dict + <i>valid</i> : Boolean = False + <i>notify_nodes</i> : Set + <i>logger</i> : logging.Logger + <i>listening_thread</i> : threading.Thread + <i>analyse_thread</i> : threading.Thread
+ <i>StateMonitor</i> ( <i>node_count</i> : Integer) - <i>_start_listening_thread</i> () - <i>_start_analyse_thread</i> () - <i>_handle_state</i> ( <i>node_state</i> : <i>SortedListNodeState</i> ) - <i>_write_result_message</i> () + <i>stop</i> ()

Abbildung 5.2: StateMonitor Klasse

Um nun zu prüfen, ob ein System von Knoten stabil ist, wird innerhalb des StateMonitors die *system\_has\_valid\_state(nodes, logger)* Funktion aufgerufen. Dabei werden die Knoten als Dictionary mit der *NodeId* als Key und dem Node Status als Wert angegeben. Außerdem wird ein Logger als Parameter gefordert, wodurch Debug Informationen geloggt werden können. Die Validierung prüft zuerst, ob jeder Knoten, der im System bleibt, auch aktiv läuft. Zweitens wird geprüft, ob jeder Knoten, der das System verlassen will, „tot“ ist und somit seinen *running* Wert auf *False* hat. Sollte ein Knoten die beiden Bedingungen nicht erfüllen wird

*False* zurückgegeben und das System als nicht stabil identifiziert. Sollten die beiden vorherigen Bedingungen erfüllt sein, wird jeder Knoten darauf überprüft, ob dieser ein valider Knoten innerhalb der Sortierten Liste ist. Dafür wird die Funktion *check\_sorted\_list\_node\_is\_valid(node, node\_ids, should\_be\_connected=False)* benutzt. Der Ablauf dieser Funktion wird später näher erklärt. Sollte ein Knoten kein valider Knoten innerhalb der Sortierten Liste sein, wird *False* zurückgegeben. Andernfalls wird zum Schluss geprüft, ob alle bleibenden Knoten schwach zusammenhängend sind. Erst wenn alle Bedingungen erfüllt sind, wird *True* zurückgegeben und das System als stabil bezeichnet. Die *check\_sorted\_list\_node\_is\_valid* Funktion prüft einen Knoten dahingehend, ob dieser einen korrekten Zustand innerhalb der Sortierten Liste besitzt. Dafür wird der Knotenstatus als Parameter, alle Knoten IDs von bleibenden Knoten in sortierter Reihenfolge und eine Flag, die speichert, ob ein Knoten Verbindungen zu anderen Knoten besitzen sollte, angegeben. Diese Flag ist *False* bei Knoten, welche das System verlassen wollen, und *True* bei bleibenden Knoten. Sollte der linke und rechte Nachbar des Knotens nicht gesetzt sein und es sich um einen verlassenden Knoten handelt, wird *True* zurückgegeben, andernfalls ist der Knoten nicht valide und es wird *False* zurückgegeben. Auch wird *False* zurückgegeben, wenn ein verlassender Knoten einen linken oder rechten Nachbarn besitzt. Sollte nur einer der beiden Nachbarn gesetzt sein, muss es sich, damit der Knoten ein valider Knoten darstellt, um einen Randknoten handeln. Es wird also geprüft, wenn nur der rechte Nachbar gesetzt ist, ob die ID des Knotens die kleinste in dem System der bleibenden Knoten ist. Auch wird geprüft, ob die Verbindung zum rechten Nachbarn direkt ist und ob der rechte Nachbar wirklich der nachfolgende Knoten mit der nächsthöheren ID ist. Analog dazu wird geprüft, wenn nur der linke Nachbar gesetzt ist, ob dieser Knoten der mit der größten ID ist. Sollten beide Nachbarn gesetzt sein, wird geprüft, ob die ID des linken Nachbarn kleiner als die eigene ID und die ID des rechten Nachbarn größer als die eigene ID ist. Außerdem müssen diese Verbindungen direkt sein. Weiterhin wird in diesem Fall geprüft, ob die ID des linken Nachbarn wirklich die nächstkleinere ID und die ID des rechten Nachbarn die nächstgrößere ID ist. Wenn eine der Bedingungen falsch ist, wird *False* zurückgegeben und der Knoten ist nicht valide. Bei der Überprüfung des schwachen Zusammenhangs mit der Funktion *check\_weak\_connectivity(nodes)*, wird lediglich aus allen Relays in den Knoten eine Adjazenzmatrix erstellt und anschließend mit der *BFS(visited\_nodes, adjacency)* Funktion eine Breitensuche durchgeführt. Nachdem die Breitensuche durchgeführt wurde, wird geprüft, ob jeder Knoten einmal besucht wurde.

### 5.1.1 Zufällige Graphgenerierung

Wir wissen nun, wie wir Knoten im Analysemodus ausführen können und wie wir Systeme von Knoten mit dem StateMonitor überwachen können. Nun bleibt die Frage, wie Knotensysteme erstellt werden können, um deren Stabilisierung analysieren zu können. Dafür benötigen wir einen Graphen von Knoten, welcher durch Verbindungen von Relays einen schwachen Zusammenhang bildet. Dabei ist es irrelevant, ob der initiale Graph eine Sortierte Liste ist oder nicht. Für die Arbeit wurde sich dazu entschlossen zufällig falsche Graphkonfigurationen als initialen Graphen zu benutzen, um daran zu analysieren, wie gut die Implementierung das System stabilisiert und Knoten das System verlassen. Für die Generation dieser zufälligen Startkonfigurationen wurden einige Klassen und Funktionen in der *GraphGeneration.py* Datei erstellt, welche nachfolgend kurz erläutert werden.

Da wir in der Analyse jeden Knoten in einem eigenen Prozess starten, wird eine Möglichkeit benötigt, Verbindungen zu anderen Knoten zu speichern, um diese später bei der Erstellung des Knotens einzurichten. Dafür wurde die *Blueprint* Klasse erstellt, welche in Abbildung 5.3 zu sehen ist. Diese Klasse speichert eingehende und ausgehende Verbindungen eines Knotens. Die wichtigen Informationen für eingehende Verbindungen zu einem Knoten werden in der Klas-

se *InBlueprint* gespeichert. In dieser Klasse wird der Key, welcher die eingehende Verbindung authentifiziert, und die RelayLayerId gespeichert, welche den **RelayLayer** angibt, von dem die Verbindung ausgeht. Alle ausgehenden Verbindungen werden in der *OutBlueprint* Klasse gespeichert. Diese Klasse speichert den Key, welcher für die ausgehende Verbindung benötigt wird, die RelayLayerRid des Sink-Relays und eine *direction* Information. Die *direction* Information gibt an, ob die ausgehende Verbindung als linker oder rechter Nachbar gesetzt werden soll. Bei dem Wert 0 wird die ausgehende Verbindung nicht als Nachbar gesetzt, bei 1 als linker und bei 2 als rechter Nachbar. Innerhalb der *Blueprint* Klasse werden nun alle eingehenden und ausgehenden Blueprints in einer Liste gespeichert. Somit kann jeder Knoten beim Erhalt eines *Blueprint* Objekts alle Verbindungen erstellen. Wenn wir nun einen initialen Graphen erstellen wollen, müssen wir pro Kante von einem Knoten  $v$  nach  $u$  ein *InBlueprint* Objekt in das *Blueprint* Objekt von  $u$  einfügen und ein *OutBlueprint* Objekt in das *Blueprint* Objekt von  $v$  einfügen.

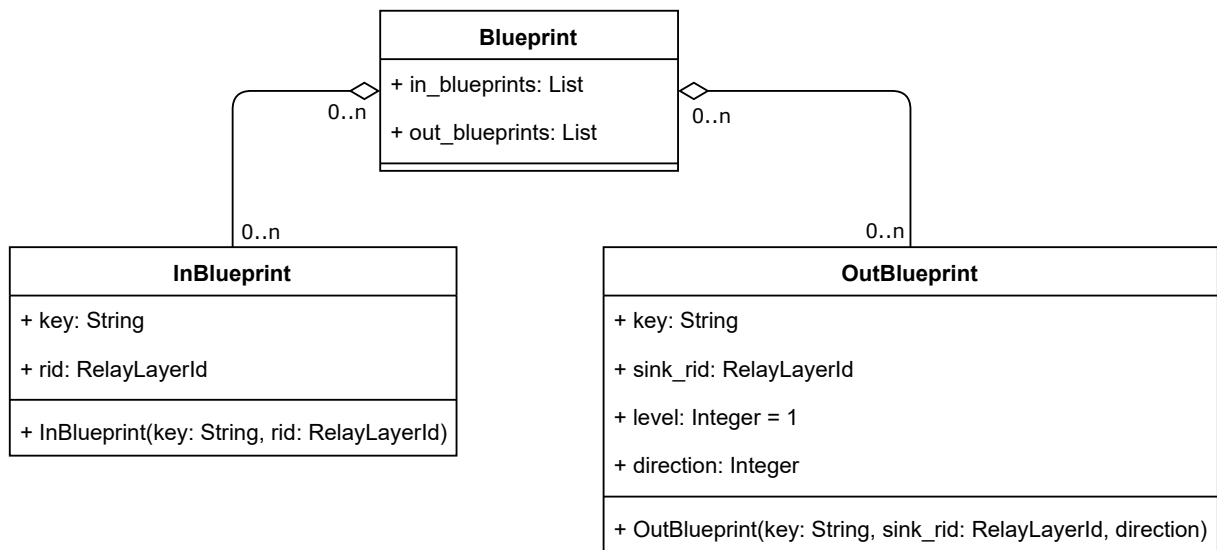


Abbildung 5.3: Blueprintklassen zur Generierung von Graphen

Um nun einen Graphen zufällig zu erstellen, wird der Algorithmus 6 vorgestellt. Innerhalb des Pseudocodes wird jedoch die genaue Implementierung der Verbindungen von zwei Knoten nicht aufgeführt. Diese Verbindung wird mithilfe der oben beschriebenen Blueprints gespeichert. Das grundlegende Prinzip des Algorithmus ist, bereits gebildete Zusammenhangskomponenten miteinander zu verbinden, damit am Ende eine einzelne Zusammenhangskomponente entsteht.

**Satz 5.1.** *MakeRandomGraph bildet einen Graphen der schwach zusammenhängend ist.*

*Beweis.* Am Anfang der Funktion wird jeder angegebene Knoten in einer eigenen Gruppe gespeichert. Da es noch keine Kanten zwischen den Gruppen gibt, ist jeder Knoten eine eigene Zusammenhangskomponente. Solange es mehr als eine Gruppe gibt, werden zufällig zwei Zusammenhangskomponenten  $G$  und  $G'$  ausgewählt. Diese zwei Zusammenhangskomponenten werden dann mit zwei zufälligen Knoten innerhalb der Zusammenhangskomponente verbunden. Daraus folgt, dass es einen Pfad von einem Knoten aus  $G$  zu einem Knoten aus  $G'$  gibt. Daraus folgt wiederum, dass der ungerichtete Graph  $\mathcal{G}(D)$  aus dem gerichteten Graph  $D = G \cup G'$  zusammenhängend ist. Somit ist der gerichtete Graph der beiden Gruppen schwach zusammenhängend. Nun können wir also die beiden Gruppen als eine Zusammenhangskomponente zusammenfügen. Das beschriebene Prozedere wird dann weiter durchgeführt. Sollten irgendwann nur noch zwei Zusammenhangskomponenten existieren, werden diese, wie oben beschrieben, wiederum miteinander verbunden und zusammengefügt. Diese ist ebenfalls schwach zusammenhängend,

**Algorithm 6** MakeRandomGraph

---

```

1: function MAKERANDOMGRAPH(nodes)
2:   Speichere jeden Knoten aus nodes in einer eigenen Gruppe
3:   while Länge der Gruppen > 1 do
4:     group1  $\leftarrow$  Wähle zufällig eine Gruppe von Knoten aus
5:     group2  $\leftarrow$  Wähle zufällig eine Gruppe von Knoten aus mit group1  $\neq$  group2
6:     node1  $\leftarrow$  Wähle zufällig einen Knoten aus group1
7:     node2  $\leftarrow$  Wähle zufällig einen Knoten aus group2
8:     direction  $\leftarrow$  Zufällige Zahl aus  $\{0, 1\}$ 
9:     if direction = 0 then
10:       Erstelle Kante von node1 zu node2
11:     else
12:       Erstelle Kante von node2 zu node1
13:     end if
14:     Füge group1 und group2 zusammen
15:   end while
16:   return Die einzige übrig gebliebene Gruppe
17: end function

```

---

da zwei zuvor schwach zusammenhängende Komponenten miteinander verbunden wurden. Da dann nur eine Zusammenhangskomponente existiert, terminiert der Algorithmus und es wird diese Zusammenhangskomponente zurückgegeben, welche einen schwach zusammenhängenden gerichteten Graphen repräsentiert.  $\square$

Die Funktion *make\_weakly\_connected\_sorted\_list* arbeitet wie der oben beschriebene Algorithmus. Jedoch speichert die Funktion den Graphen innerhalb von Blueprints. Diese Blueprints sind der Wert innerhalb eines Dictionaries und besitzen als Key die RelayLayerId eines Knotens. Weiterhin wird bei der Verbindung von zwei Knoten zufällig ausgewählt, ob diese Verbindung als ein Nachbar in dem Knoten gesetzt werden soll. Die Benutzung dieser Blueprints und Einrichtung von Verbindungen zwischen Knoten kann beispielhaft in der main.py Datei eingesehen werden. Mit der main.py Datei kann eine Analyse durchgeführt werden. Dabei kann als Argument die Anzahl an Knoten angegeben werden. Die Datei erstellt einen Graphen mit der Anzahl an Knoten, startet diese Knoten innerhalb eigener Prozesse, startet einen StateMonitor und wartet bis das System stabil ist. Anschließend werden alle Knoten heruntergefahren und das Script beendet. Damit analysiert werden kann, wie schnell die Knoten das System verlassen können, benötigen wir Knoten, die das System verlassen wollen. Beim Erstellen eines Knotens besitzt jeder Knoten in der Analyse eine 25-prozentige Chance das System verlassen zu wollen. Wenn der Knoten das System verlassen möchte, wird lediglich die *stop* Methode des Knotens aufgerufen, welche den *leaving* Wert auf *True* setzt. Das Protokoll des Knoten verlässt dann eigenständig das System.

Nun wurden alle Voraussetzungen für eine Analyse zusammengetragen, sodass die Auswahl der Analysemethode erläutert werden kann.

### 5.1.2 Methodik

Insgesamt wurden 900 Simulationen von zufälligen Graphen durchgeführt. Dabei wurden neun verschiedene Knotenmengen mit jeweils 100 Simulationen durchgeführt. Die kleinste Knotenkonfiguration besteht aus fünf Knoten und die größte aus 45 Knoten. Die Schrittweite zwischen den unterschiedlichen Mengen beläuft sich auf fünf Knoten. Die Wahrscheinlichkeit, dass ein Kno-



ten das System verlassen möchte, lag bei allen Simulationen bei 25%. Alle Analyseergebnisse sind in dem *results* Ordner gespeichert und können mit der *resultAnalysis.py* Datei ausgewertet und dargestellt werden. Die Ergebnisse einer Simulation werden als serialisiertes Objekt gespeichert und bestehen aus einer Liste mit zwei Inhalten. Der erste Inhalt ist ein Dictionary des StateMonitors, welches die Knotenstatus beinhaltet. Dabei ist der Key eines Wertes die Knoten ID und der Wert ein SortedListNodeState Objekt. Der zweite Inhalt der Liste besteht aus den Informationen über die Timeouts der einzelnen Knoten. Diese ist wiederum ein Dictionary mit der Knoten ID als Key und einer Ganzzahl als Wert. Diese Werte speichern die Anzahl an Timeouts, welche ein bestimmter Knoten durchgeführt hat. Innerhalb der *resultAnalysis.py* Datei werden alle Ergebnisdateien, welche sich in dem Ordner befinden, zusammengefasst und entsprechend aufgearbeitet. Um den Einlesevorgang bei jeder Durchführung zu vereinfachen, werden Zwischendateien geschrieben. Diese Dateien werden für die Knotendaten und die Timeoutdaten geschrieben. Sollten neue Ergebnisdateien analysiert werden, müssen zuerst diese Zwischendateien gelöscht werden, damit neue Daten in die Ergebnisse fließen können.

Generell wurde die Analyse der Ergebnisse in zwei Teile aufgeteilt. Der erste Teil untersucht die maximalen Timeouts jeder Simulation bis das System stabilisiert wurde. Der zweite Teil bezieht sich auf die Timeouts der Knoten, welche das System verlassen wollen. Um die beiden Teile betrachten zu können, werden drei Diagramme erstellt. Alle Diagramme sind durch Boxplots dargestellt, wobei anhand der einzelnen Mengen an Knoten unterschieden wird. Beim ersten Diagramm wird für jede Simulation die maximale Anzahl an Timeouts zusammengetragen. Aus diesen maximalen Timeouts werden anschließend Boxplots gebildet, welche die Anzahl der Timeouts anzeigen bis das System stabilisiert wurde. Die beiden nächsten Diagramme beziehen sich auf die Timeouts der verlassenden Knoten. Für das zweite Diagramm werden alle Timeouts von diesen Knoten zusammengetragen und daraus der Durchschnitt gebildet. Somit zeigt das Diagramm, wie viele Timeouts ein Knoten durchschnittlich in einer Simulation gebraucht hat, um das System zu verlassen. Das letzte Diagramm zeigt hingegen die maximalen Timeouts pro Simulation, welche gebraucht werden, damit alle Knoten das System erfolgreich verlassen konnten.

## 5.2 Analyseergebnisse

Nachfolgend werden zuerst die benötigten Timeouts zur Stabilisierung des Systems und anschließend die benötigten Timeouts zum Verlassen des Systems dargestellt. Dabei werden die wichtigsten Werte näher erläutert. Zum Schluss werden die Ergebnisse kritisch bewertet und diskutiert.

### 5.2.1 Stabilisierung des Systems

In der Abbildung 5.4 ist die Anzahl der Timeouts, welche benötigt wurden, um das System bei einer Simulation zu stabilisieren, in Bezug auf die Anzahl an Knoten pro Simulation als Boxplot dargestellt. Deutlich zu sehen ist, dass der Durchschnitt, welcher durch den dunkelroten Punkt dargestellt wird, sich bei allen Simulationen um die zehn Timeouts befindet. Weiterhin steigt dieser bei steigender Anzahl von Knoten nahezu linear an. Die Ausnahmen dieses Anstiegs bestehen in der Simulation von zehn und 45 Knoten, bei welcher der Durchschnitt wesentlich höher liegt als ein linearer Anstieg vermuten ließe. Weiterhin liegen die Mediane und die Durchschnitte der Simulationen bei den meisten Knotenmengen sehr nah beieinander oder sogar aufeinander. Lediglich die Simulation mit zehn, 35 und 45 Knoten bilden hier die Ausnahme. Ein wesentlicher Punkt ist, dass die Ausreißer nach oben bei ansteigender Knotenmenge sehr stark zunehmen. Zwischen fünf und 20 Knoten sind die größten Ausreißer um die 15 Timeouts gelegen. Bei 45

Knoten hingegen liegt der größte Ausreißer bei 44 Timeouts. Generell sind bei den meisten Simulationen 50% der Werte nah beieinander liegend. Lediglich die Simulation von zehn Knoten und von 45 Knoten bildet dort eine Ausnahme. Bei den Simulationen von 25 und 35 Knoten sind 50% der Werte am nächsten gelegen. Die niedrigsten Werte der Simulationen steigen ebenso linear an. Der niedrigste Wert bei der Simulation von fünf Knoten liegt bei drei Timeouts. Ab zehn bis 20 Knoten steigt dieser auf fünf Timeouts. Von 25 bis 30 Knoten liegt der niedrigste Wert bei sechs Timeouts. Die Simulation von 35 Knoten ist dagegen die einzige Simulation mit sieben Timeouts als niedrigsten Wert. Ab 40 Knoten liegt der niedrigste Wert bei acht Timeouts.

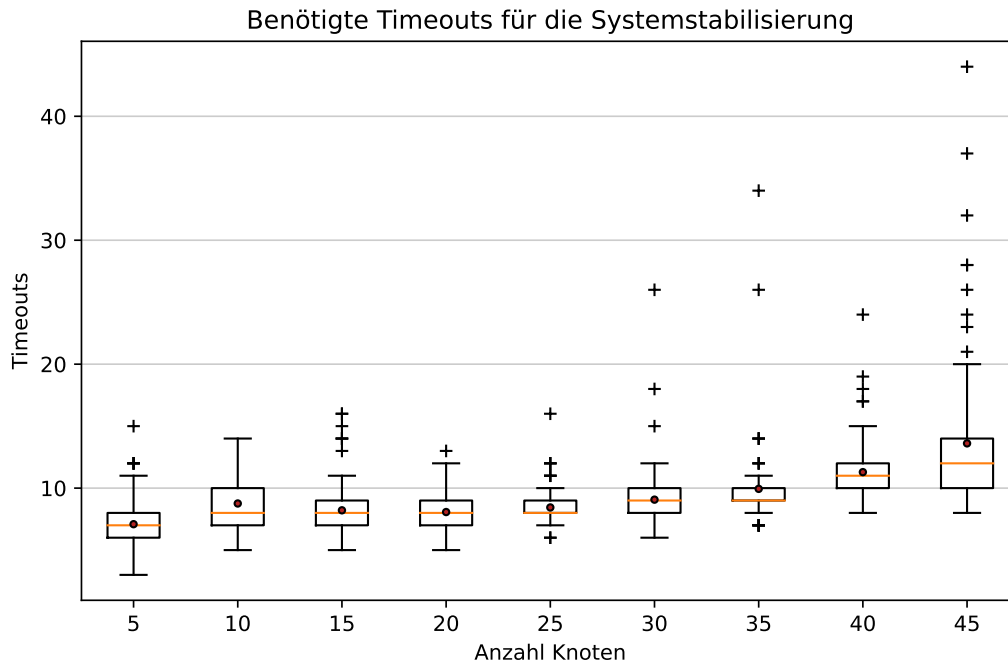


Abbildung 5.4: Ergebnisse für Stabilisierung

### 5.2.2 Verlassen des Systems

In Abbildung 5.5 sind die beiden Diagramme gegeben, welche die Werte der Knoten, die das System verlassen, näher zeigen. Das erste Diagramm betrachtet die durchschnittlichen Timeouts dieser Knoten bei jeder Simulation. Das zweite hingegen zeigt die maximalen Timeouts. Dabei ist zu erkennen, dass die niedrigsten Werte aller Simulationen nicht kleiner als zwei sind. Bei den Simulationen von fünf bis 35 Knoten liegen diese sogar genau bei zwei. Lediglich bei den Simulationen von 40 bis 45 Knoten steigen diese Werte leicht an. Zuerst gehen wir auf die Ergebnisse des ersten Diagramms ein. Bei den Durchschnittstimeouts ist zu sehen, dass die Durchschnitte und Mediane der Simulationen von 15 bis 45 Knoten zwischen drei und fünf Timeouts liegen. Lediglich die Simulationen mit fünf und zehn Knoten besitzen Durchschnitte von über fünf Timeouts bei zehn Knoten und über sechs Timeouts bei fünf Knoten. Ab 30 Knoten ist ein linearer Aufwärtstrend in den Medianen und Durchschnitten zu sehen. Weiterhin ist hier zu erkennen, dass ab 15 Knoten 50% der Werte sehr nah beieinander liegen. Bei den Knotenmengen fünf bis 15 hingegen liegen 50% der Werte weiter auseinander als bei den folgenden Simulationen. Auch die Durchschnitte der Simulationen mit fünf bis 15 Knoten liegen höher als die der nachfolgenden Knotenmengen und ein Abwärtstrend von fünf bis 25 Knoten ist zu erkennen. Auch die Streuung

der Werte ist bei fünf Knoten am größten und nimmt bis 25 Knoten ab. In den Ausreißern ist kein wesentlicher Trend zu entdecken. Jedoch ist der größte Ausreißer mit 14 Timeouts bei einer Knotenmenge von fünf zu beobachten.

Bei dem zweiten Diagramm, welches die maximalen Timeouts beim Verlassen des Systems betrachtet, liegen bei allen Simulationen 50% der Timeouts unter zehn. Die Durchschnitte und Mediane der Simulationen liegen auch alle zwischen fünf und 10 Timeouts und liegen nah beieinander. Weiterhin sind die mittleren 50% der Timeouts nicht weiter als fünf Timeouts voneinander entfernt. Ein wesentlicher Punkt ist, dass die Ausreißer nach oben bei steigender Knotenmenge zunehmen. So sind die Ausreißer bei fünf bis 25 Knoten gleich oder unter 15 Timeouts. Ab 30 Knoten steigen die Ausreißer über 15 Timeouts. Der größte Ausreißer ist bei 35 Knoten mit 34 Timeouts zu beobachten. Die meisten Ausreißer hingegen befinden sich bei 45 Knoten. Generell befinden sich die Werte für jede Knotenmenge gleichmäßig auf einem Level. Abgesehen von den Ausreißern ist kein Trend in eine bestimmte Richtung zu erkennen.

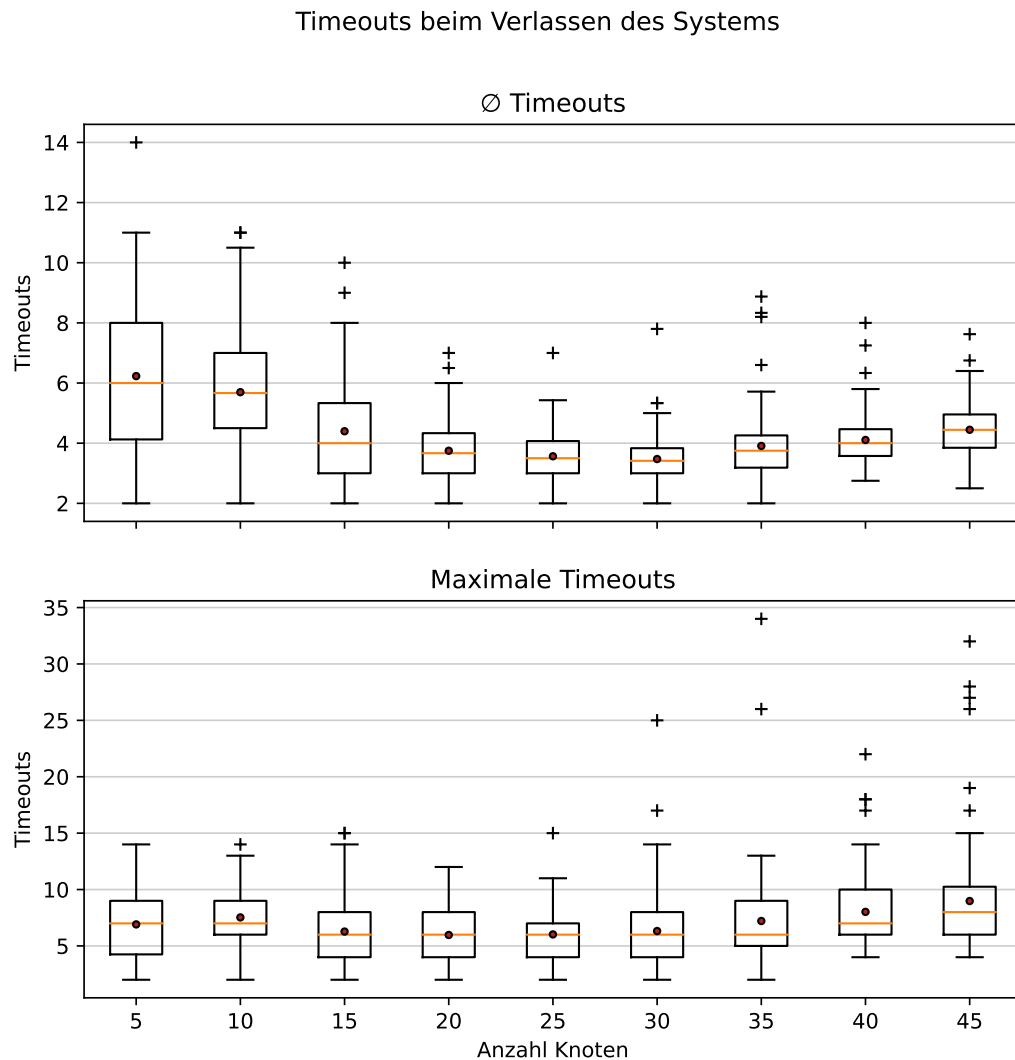


Abbildung 5.5: Ergebnisse für das Verlassen des Systems

### 5.2.3 Einordnung der Ergebnisse

Generell kann man aus den Ergebnissen schließen, dass bei höherer Knotenanzahl auch die benötigten Timeouts zur Stabilisierung des Systems ansteigen. Dieses Verhalten ist zu erwarten, da mehr Knoten durch das BuildList Protokoll linearisiert werden müssen. Auch die signifikant hohen Ausreißer bei erhöhter Knotenmenge sind zu erwarten, da bei steigender Knotenanzahl auch häufiger zufällige Konfigurationen auftreten, die mehr *Linearize* Aktionen und Timeouts benötigen. Außerdem ist die Menge an Nachrichten, die übermittelt werden, größer. Da die Knoten nicht synchron arbeiten, kann es passieren, dass Nachrichten in Knoten länger für das Verarbeiten benötigen und somit mehr Timeouts durchgeführt werden als theoretisch benötigt werden. Nichtsdestotrotz weisen die Durchschnitte der Stabilisierung, welche nie größer als 15 Timeouts sind, darauf hin, dass die Stabilisierung mit dem Relay-Modell effizient funktioniert. Bei den Ergebnissen für das *FDP* ist aufgefallen, dass die minimalen Timeouts nicht kleiner als zwei sind. Dies ist darauf zurückzuführen, dass beim Protokoll für das Verlassen eines Knotens zuerst alle Relays in einem Timeout gelöscht werden, die eigenen Verbindungen versucht werden umzukehren und ein Anker gesucht wird. Erst wenn der eigene Sink-Relay keine eingehenden Verbindungen und keine Relays mehr besitzt, kann der Knoten das System verlassen. Jedoch können die eingehenden Verbindungen erst entfernt werden, wenn alle Knoten gefragt wurden, diese Verbindung umzukehren. Dieses Vorgehen dauert mindestens zwei Timeouts und erklärt die untere Grenze. Ein weiterer Aspekt, der aufgefallen ist, ist dass die durchschnittlichen Timeouts beim Verlassen des Systems für kleine Knotenmengen sehr stark abweichen und sogar bei fünf Knoten den größten Ausreißer vorweisen. Dieses Verhalten ist damit zu erklären, dass nicht immer gleich viele Knoten das System verlassen, sondern, wie in Abschnitt 5.1 erläutert, die Wahrscheinlichkeit für das Verlassen bei 25% liegt. Daraus kann folgen, dass nur sehr wenige oder nur ein Knoten das System verlassen möchte. Wenn nun dieser Knoten durch bestimmte Gegebenheiten das System erst sehr spät verlassen kann, können diese Werte nicht durch andere Knoten ausgeglichen werden, wie es bei mehreren Knoten der Fall wäre. Beim Implementieren ist aufgefallen, dass der Grund für die lange Wartezeit eines Knotens durch die Überprüfung der eingehenden Verbindungen des Sink-Relays gegeben ist. Ein Knoten kann nur verlassen, wenn diese eingehenden Verbindungen gelöscht sind. Er muss also auf den Moment warten, in dem der *RelayLayer* eine *InRelayClosed* Nachricht bekommt, um die Verbindung zu trennen. Sollte in der Zwischenzeit aber wiederum durch das Protokoll die Referenz zu dem Sink-Relay an einen anderen Knoten geschickt werden, existiert wieder eine eingehende Verbindung, die gelöscht werden muss.

Bei der Betrachtung der maximalen Timeouts, die repräsentieren, wie lange ein System gebraucht hat bis alle verlassenden Knoten das System verlassen haben, ist aufgefallen, dass die Ausreißer bei großen Knotenmengen größer ausfallen und häufiger auftreten. Dies kann entweder durch die erhöhte Nachrichtenrate oder durch die Bildung von Relayketten erklärbar sein, welche teilweise sehr große Level besitzen. Die langen Ketten müssen zunächst zu direkten Verbindungen umgewandelt werden, damit diese durch das Protokoll gelöscht werden können. Erst wenn alle Relays gelöscht wurden, kann der Knoten das System verlassen. Dies verzögert sich demnach durch die Weiterleitung von Relays.

Weiterhin sind bei den maximalen Timeouts bei der Simulation von 35 Knoten zwei Ausreißer zu erkennen. Diese beiden Ausreißer sind auch in Abbildung 5.4 wiederzufinden. Somit sind die Ausreißer beim Stabilisieren des Systems dadurch zu erklären, dass das System auf das Verlassen eines oder mehrerer Knoten gewartet hat, bevor es als stabil angesehen wurde. Auch einige Ausreißer der maximalen Timeouts aus anderen Simulationen sind in der Abbildung 5.4 wiederzufinden. Somit können wir sagen, dass die Ausreißer beim Stabilisieren des Systems häufig vom Verlassen eines oder mehrerer Knoten abhängen.

Trotzdem können wir feststellen, dass sowohl bei den durchschnittlichen Timeouts als auch bei

den maximalen Timeouts die mittleren Werte nie über zehn Timeouts liegen. Das bedeutet, dass das Modell in relativ kurzer Zeit Knoten aus dem System sicher entfernen kann. Weiterhin muss betrachtet werden, dass die Simulation keine stabilen Systeme betrachtet und in diesen analysiert, wie schnell Knoten das System verlassen können. Stattdessen werden Systeme analysiert, welche im Prozess des Verlassens zeitgleich stabilisiert werden. Die Stabilisierung eines Systems ist zudem von der Stabilisierung der **RelayLayer** abhängig. Zum Beispiel ist das Verlassen eines Knotens davon abhängig, wie schnell die **RelayLayer** auf Löschanfragen reagieren und eingehende Verbindungen löschen. Ein wesentlicher Punkt, der die Simulation verzögert, ist die Aufstauung von Nachrichten in den Nachrichten Buffern. Die beschriebene Simulation wurde mit der Option durchgeführt, dass nicht erreichte **LinkLayer** keine Closed Nachricht auslösen. Dadurch kann es passieren, dass lange auf das erfolgreiche Senden einiger Nachrichten gewartet wird, obwohl die **LinkLayer** nicht erreichbar sind. Somit konnten wichtige Nachrichten nicht direkt an die entsprechenden **RelayLayer** gesendet werden und die Stabilisierung des Systems verzögerte sich.

#### 5.2.4 Erwähnenswerte Beobachtungen

Bei der Implementierung der Simulation und der Library kam es zu relevanten Beobachtungen, die im folgenden erläutert werden. Zum einen werden innerhalb des Protokolls Abfragen sowohl über die Keys als auch über die eingehenden Referenzen gefordert. Die Implementierungen dieser Abfragen iterieren über alle eingehenden Referenzen, um zum Beispiel zu überprüfen, ob ein bestimmter Key innerhalb einer eingehenden Referenz vorhanden ist. Da bei den zugrundeliegenden Protokollen die **Self-introduction Annahme** (siehe Abschnitt 2.4) gilt, werden häufig die eigenen Sink-Relays anderen Knoten vorgestellt. Durch die Vorstellung des immer gleichen Relays wird im Sink-Relay bei jeder Vorstellung eine neue eingehende Referenz erstellt und im empfangenden Knoten die ausgehenden Informationen gemerged. Dabei entstehen nach einiger Zeit sehr viele eingehende Referenzen und ausgehende Keys bei einigen Relays. Nach sehr langer Laufzeit kann die Menge der Referenzen zu verlängerten Laufzeiten von Methoden führen.

Weiterhin können Verzögerungen von Methoden dazu führen, dass Nachrichten, welche sich in dem Nachrichtenbuffer des **RelayLayer** befinden, sich nach einiger Zeit aufstauen und so das gesamte System verlangsamen. In Anbetracht dessen, dass bei Overlay-Netzwerken die Ausführungszeit von geringer Bedeutung ist und Verzögerungen auftreten dürfen, ist diese Problematik nicht von großer Relevanz. Jedoch ist dieses Problem bei der Simulation negativ aufgefallen und führte zu einigen Neustrukturierungen des Quellcodes im Verlauf dieser wissenschaftlichen Arbeit. Diese Verzögerungen der Methoden könnte auch dazu geführt haben, dass einige Ausreißer bei den Simulationen aufgetreten sind.



## Ausblick und Zusammenfassung

In diesem Kapitel werden wir zuerst offene Fragestellungen vorstellen. Anschließend wird die Arbeit zusammengefasst und die wichtigsten Ergebnisse werden noch einmal dargestellt.

### 6.1 Ausblick

In den Referenzen [Set20, SS18] wurde darauf hingewiesen, dass der Aufbau von Relayverbindungen noch nicht definiert wurde. Demnach ist es also aktuell noch nicht möglich eine Verbindung von einem Knoten zu einem anderen Knoten aufzubauen, wenn diese Verbindung noch nicht zuvor existiert. In dieser Arbeit wurden die Verbindungen vor der Simulation explizit aufgebaut, sodass der zugrundeliegende Systemgraph schwach zusammenhängend ist. Es bleibt somit offen, wie Verbindungen zu anderen Knoten innerhalb eines Protokolls aufgebaut werden können.

Das originale Relay-Modell schlägt vor, dass der Relay-Layer und der Link-Layer abgesondert von dem eigentlichen Knoten implementiert werden soll. Dadurch soll es nicht möglich sein von außerhalb auf die Relays und Attribute des Relay-Layers zuzugreifen. Da die zeitliche Beschränkung dieser Arbeit es nicht zugelassen hat, den Relay-Layer abgesondert zu implementieren, bleibt die Umsetzung der Implementierung offen. Es müssten Möglichkeiten entwickelt werden, den Relay-Layer innerhalb einer abgeschotteten „Umgebung“ laufen zu lassen. Dafür werden Schnittstellen benötigt, über die der Knoten Methoden aufrufen kann. Ob diese Abgrenzung mittels Hardware oder Software geschieht, bleibt noch zu studieren.

Voraussetzung für das Funktionieren des Relay-Modells ist, dass die Nachrichten, welche an einem Relay-Layer ankommen, valide Nachrichten sind. Es sollte also nicht möglich sein, falsche Nachrichten an ein Relay-Layer zu senden. Da die Verbindung in den `LinkLayer` über ungesicherte Sockets funktioniert, könnten arbiträre Nachrichten an ein `LinkLayer` geschickt werden. Um das zu verhindern, sollten Nachrichten, welche über den `LinkLayer` geschickt werden, gesichert werden. Daher sollten vor allem Sicherungen der Vertraulichkeit und Integrität der Nachrichten entwickelt werden. Die Integrität sollte geschützt werden, damit zum Beispiel keine Relays von außerhalb gelöscht werden können. Die Vertraulichkeit muss gegeben sein, damit keine Keys von Übertragungen abgefangen werden können. Sollten Keys abgefangen werden, könnte man Nachrichten über ein `RelayLayer` senden, ohne dass die Erlaubnis dafür vorher gegeben wurde. Die Sicherung der übertragenen Daten bleibt also offen und könnte in späteren Arbeiten erweitert werden.

In dieser Arbeit wurden die DoS Erweiterungen lediglich vorgestellt und implementiert. Wie gut diese Erweiterungen vor DDoS Angriffe schützen können, wurde nicht studiert. Somit könnten

in weiteren Arbeiten Praxisanalysen durchgeführt werden, welche die Güte der Erweiterungen analysiert. Weiterhin können Analysen darüber geführt werden, welche Einstellungen der Konfiguration am besten vor diesen Angriffen schützen können, damit zum Beispiel falsch-positive Erkennungen klein gehalten werden. Außerdem könnten eventuell bessere Methoden zur Erkennung entwickelt werden als die Erweiterung dieser Arbeit. Es könnte zum Beispiel der Erkennungsalgorithmus von einer Fenstermethode auf den CUSUM Algorithmus [BN93] abgeändert und analysiert werden.

Die Simulationen wurden alle mit einer einzigen Einstellung des Moduls durchgeführt. Es wäre interessant zu sehen, wie unterschiedlich Simulationen verlaufen, wenn die Konfigurationen von Timeoutzeiten unterschiedlich gewählt werden.

Ein negativer Punkt, der bei der Implementierung aufgefallen ist, ist, dass eingehende Referenzen und Keys sich bei langer Ausführungszeit anhäufen. Damit die Verzögerung von Methoden nicht enorm vergrößert wird, könnte man die Implementierung der Library so anpassen, dass diese bessere Datenstrukturen benutzt, um Abfragen der Keys schneller zu gestalten. Eine andere Möglichkeit wäre, das Relay-Modell so anzupassen, dass die Menge an eingehenden Referenzen oder Keys von den gleichen Relays klein gehalten wird oder reduziert werden kann.

## 6.2 Zusammenfassung

In dieser Arbeit wurde das von Scheideler und Setzer vorgestellte Relay-Modell [SS18, Set20] in einer nutzbaren Library implementiert. Diese Library wurde in Python programmiert und die Implementierung in dieser Arbeit erläutert. Des Weiteren wurden Erweiterungen für das Modell entwickelt, welche DDoS Angriffe erkennen und verhindern können. Voraussetzung dafür ist jedoch, dass alle Verbindungen von unterschiedlichen Knoten über Relays laufen, welche durch den Relay-Layer verwaltet werden. Da das Modell entwickelt wurde, um das  $\mathcal{FDP}$  von Overlay Netzwerke zu lösen, wurde zudem in dieser Arbeit anhand der implementierten Library eine Praxisanalyse des Problems durchgeführt. Dafür wurde das BuildList Protokoll [FSS20] so implementiert, dass die implementierte Library des Relay-Modells verwendet wird. Für die Analyse wurden verschiedene Knotenmengen simuliert, welche zufällig falsche Graphenkonfigurationen als initialen Systemgraph besaßen. Die einzige Voraussetzung der Konfiguration ist, dass diese einen schwach zusammenhängenden Graph formt, sodass das BuildList Protokoll [FSS20] eine Sortierte Liste mit allen Knoten bilden kann. Außerdem besaß jeder Knoten in den Simulationen eine Wahrscheinlichkeit von 25% das System verlassen zu wollen. Wir kamen zu dem Ergebnis, dass die Knoten in kurzer Zeit die Systeme verlassen haben. Dabei wurde die Zeit, welche ein Knoten braucht bis er das System verlässt, in der Anzahl von ausgeführten Timeouts gemessen. Weiterhin ist aufgefallen, dass die minimale Anzahl an Timeouts für das Verlassen des Systems bei zwei Timeouts liegt. Des Weiteren ist die durchschnittliche Timeoutanzahl bis alle Knoten das System verlassen haben, bei allen Simulationen zwischen fünf und zehn Timeouts. Bei größerer Knotenmenge ist die Wahrscheinlichkeit jedoch höher, dass ein Knoten wesentlich länger benötigt das System verlassen zu können als bei kleineren Knotenmengen. Die implementierte Library hat sich im Rahmen dieser Arbeit als erfolgreich erwiesen und gezeigt, dass das  $\mathcal{FDP}$  in der Praxis von dem Modell effizient gelöst werden kann.



# Literaturverzeichnis

- [Ben20] Viktor Bengs. Data Mining: Data Mining auf Datenströmen. Universität Paderborn, 2020.
- [BN93] Michèle Basseville and Igor Vladimirovič Nikiforov. *Detection of abrupt changes: Theory and application*. Prentice Hall information and system sciences series. Prentice-Hall, Englewood Cliffs, NJ, 1993.
- [Cis20] Cisco. Cisco Annual Internet Report (2018–2023) White Paper. <https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html>, 2020. Accessed on 2021.09.18.
- [CKBR06] Glenn Carl, George Kesidis, Richard R. Brooks, and Suresh Rai. Denial-of-service attack-detection techniques. *IEEE Internet Comput.*, 10(1):82–89, 2006.
- [Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.
- [DM03] Christos Douligeris and Aikaterini Mitrokotsa. DDoS attacks and defense mechanisms: a classification. In *Proceedings of the 3rd IEEE International Symposium on Signal Processing and Information Technology (IEEE Cat. No.03EX795)*, pages 190–193. IEEE, 2003.
- [FKN<sup>+</sup>14] Dianne Foreback, Andreas Koutsopoulos, Mikhail Nesterenko, Christian Scheideler, and Thim Strothmann. On stabilizing departures in overlay networks. In *Stabilization, Safety, and Security of Distributed Systems - 16th International Symposium, SSS 2014, Paderborn, Germany, September 28 - October 1, 2014. Proceedings*, volume 8756 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2014.
- [Fou] Python Software Foundation. uuid - uuid objects according to rfc 4122. <https://docs.python.org/3/library/uuid.html>. Accessed on 2021.09.01.
- [FSS20] Michael Feldmann, Christian Scheideler, and Stefan Schmid. Survey on algorithms for self-stabilizing overlay networks. *ACM Comput. Surv.*, 53(4):74:1–74:24, 2020.
- [GJM12] Brij B. Gupta, Ramesh Chandra Joshi, and Manoj Misra. Distributed denial of service prevention techniques. *CoRR*, abs/1208.3557, 2012.
- [Hin13] Pieter Hintjens. *ZeroMQ: messaging for many applications*. O’Reilly Media, Inc., Sebastopol, Calif., 1. edition, 2013.

- [KBG04] Daniel Kifer, Shai Ben-David, and Johannes Gehrke. Detecting change in data streams. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, pages 180–191. Morgan Kaufmann, 2004.
- [LMS05] Paul J. Leach, Michael Mealling, and Rich Salz. A universally unique identifier (UUID) URN namespace. *RFC*, 4122:1–32, 2005.
- [LRST00] Felix Lau, Stuart H. Rubin, Michael H. Smith, and Ljiljana Trajkovic. Distributed denial of service attacks. In *Proceedings of the IEEE International Conference on Systems, Man & Cybernetics: "Cybernetics Evolving to Systems, Humans, Organizations, and their Complex Interactions", Sheraton Music City Hotel, Nashville, Tennessee, USA, 8-11 October 2000*, pages 2275–2280. IEEE, 2000.
- [Sch20] Christian Scheideler. Verteilte Algorithmen und Datenstrukturen: Grundlagen. Universität Paderborn, 2020.
- [Set20] Alexander Setzer. *Local graph transformation primitives for some basic problems in Overlay networks*. PhD thesis, University of Paderborn, Germany, 2020.
- [SS18] Christian Scheideler and Alexander Setzer. Relays: A new approach for the finite departure problem in overlay networks. In *Stabilization, Safety, and Security of Distributed Systems - 20th International Symposium, SSS 2018, Tokyo, Japan, November 4-7, 2018, Proceedings*, volume 11201 of *Lecture Notes in Computer Science*, pages 239–253. Springer, 2018.
- [Wes01] Douglas B. West. *Introduction to graph theory*. Prentice Hall, Upper Saddle River, NJ, 2. edition, 2001.