Solving the Challenges of Creating a Practical Anonymous Communication System

Master-Thesis von Thilo Molitor
Tag der Einreichung: 17.09.2018

1. Gutachten: Prof. Dr. Max Mühlhäuser
2. Gutachten: Tim Grube

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Telecooperation Group

Fachbereich Informatik

Solving the Challenges of Creating a Practical Anonymous Communication System

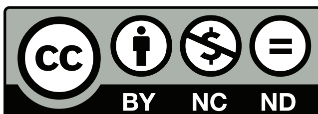
Vorgelegte Master-Thesis von Thilo Molitor

1. Gutachten: Prof. Dr. Max Mühlhäuser
2. Gutachten: Tim Grube

Tag der Einreichung: 17.09.2018

Bitte zitieren Sie dieses Dokument als:
URN: urn:nbn:de:tuda-tuprints-76853
URL: http://tuprints.ulb.tu-darmstadt.de/7685

Dieses Dokument wird bereitgestellt von tuprints,
E-Publishing-Service der TU Darmstadt
http://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de



Die Veröffentlichung steht unter folgender Creative Commons Lizenz:
Namensnennung – Nicht kommerziell – Keine Bearbeitungen 4.0 International
https://creativecommons.org/licenses/by-nc-nd/4.0/deed.de

Erklärung zur Master-Thesis gemäß § 22 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
In der abgegebenen Thesis stimmen die schriftliche und elektronische Fassung überein.

Darmstadt, den 30. Januar 2023

____

(Thilo Molitor)

Abstract

Peer to Peer (P2P) Netzwerke sind dezentrale Netzwerke, bei denen jeder Teilnehmer gleiche oder zumindest ähnliche Rollen und Funktionen übernehmen. Solche P2P Netzwerke werden meistens als Overlay auf einem anderen Netzwerk realisiert (beispielsweise dem Internet). Zwei im P2P Netzwerk benachbarte – also direkt miteinander verbundene – Knoten können im darunterliegenden Netzwerk, dem Underlay, weit voneinander entfernt und nur über mehrere Zwischenschritte verbunden sein.

Sind in einem solchen P2P Netzwerk nicht alle Knoten mit jeweils allen anderen Knoten verbunden (full mesh network), so muss das Routing im Overlay stattfinden, damit Nachrichten über potentielle Zwischenknoten weitergeleitet werden und ihr endgültiges Ziel erreichen können.

Soll das P2P Netzwerk auch noch anonym sein, so stellt das bestimmte Anforderungen an diesen Routingalgorithmus. Zwei neuartige bisher nur simulativ evaluierte Routingalgorithmen für P2P basiertes Publish-Subscribe sollen in dieser Arbeit daher erstmals in einem Forschungsprototypen implementiert und auf ihre Praxistauglichkeit untersucht werden. Das hashkettenbasierte Routing [vgl. 1] und die Ant Colony Optimization (ACO) [vgl. 2] werden dabei dem traditionellen Gossiping und Randomwalk gegenübergestellt.

Angereichert wird das Routing dabei durch die Implementierung weiterer ebenfalls bisher nur simulativ getesteter neuartiger Anonymisierungstechniken, die die Anonymität im gesamten P2P Netzwerk auch gegen globale Angreifer absichern sollen. Für alle Routingalgorithmen wurde hierfür Probabilistic Forwarding (PF) und Cover Traffix in fixen Untergruppen implementiert, die von Dauber et al. vorgeschlagene Methode namens Shell Game (SG) [vgl. 1, S. 47f] ist dagegen leider nicht praxistauglich und wurde daher auch nicht implementiert.

Inhaltsverzeichnis

1. Einleitung **1**

2. Grundlagen **3**

2.1. Hashes und Hashketten . . . 3
2.1.1. Hashketten . . . 3
2.2. Publish/Subscribe . . . 4
2.3. Peer to Peer . . . 7
2.4. Privatsphäre/Anonymität . . . 8
2.4.1. Anonymität . . . 8
2.4.2. Vertraulichkeit . . . 8
2.5. Anonymes Publish/Subscribe (Pub/Sub) mit P2P Netzwerken . . . 9
2.6. Angreifermodell . . . 11

3. Related Work **12**

4. Konzepte **14**

4.1. Allgemeine Konzepte . . . 14
4.1.1. Lokalisierung von Publishern und Subscribern eines Topics (Attributlokalisierung) 14
4.1.2. Management Messages . . . 15
4.2. Optimierungsziel . . . 16
4.3. Hashketten-basiertes Routing . . . 17
4.3.1. Advertisement Phase . . . 17
4.3.2. Subscription Phase . . . 21
4.4. ACO Routing . . . 21
4.4.1. Phase 1: Prepare Round . . . 22
4.4.2. Phase 2: Searching . . . 23
4.4.3. Phase 3: Returning . . . 23
4.5. Aktive Pfade bei ACO und Hashketten-basiertem Routing . . . 24
4.6. Anonymisierungstechniken . . . 25
4.6.1. Probabilistic Forwarding . . . 26
4.6.2. Shell Game . . . 26
4.6.3. Covergroups (Asymmetric Dining-Cryptographers Network) . . . 31

5. Probleme und Lösungen **32**

5.1. Hashketten-basiertes Routing von Daubert et al. . . . 32
5.1.1. Problem 1: Mehrere Publisher . . . 32
5.1.2. Problem 2: Knotenfehler eines Master-Publishers . . . 33
5.1.3. Problem 3: Mehrere Master-Publisher . . . 34
5.1.4. Problem 4: Nachträglich hinzugefügte Knoten . . . 35
5.1.5. Problem 5: Konkurrierende Advertisements . . . 37
5.1.6. Problem 6: Knotenfehler und alternative Pfade . . . 37
5.1.7. Problem 7: Count to Infinity . . . 38
5.1.8. Zusammenfassung der Änderungen am System . . . 40
5.2. ACO Routing . . . 42
5.2.1. Umstellung von rundenbasiert auf zeitbasiert . . . 42

5.2.2. Keine reale Entsprechung der Entität „Kante“ . . . 42
5.2.3. Zahl der Ameisenrunden . . . 43
5.2.4. Erzeugung von aktiven Kanten . . . 43
5.2.5. Versteckte Publisher . . . 43
5.2.6. Fluten einer Publish Message . . . 44
5.2.7. Overlay Maintenance . . . 44

6. Implementierung **46**
6.1. Überblick . . . 46
6.2. Das Framework . . . 46
6.2.1. Package *Networking* . . . 49
6.2.2. Package *Routing* . . . 54
6.2.3. Package *Utils* . . . 61
6.2.4. Package *Control* . . . 62
6.2.5. Filtersystem . . . 62
6.2.6. Kommandozeilenoberfläche und Mainloop . . . 64
6.3. Benutzeroberfläche . . . 64
6.4. Implementierung der Router . . . 66

7. Evaluation **67**
7.1. Framework . . . 67
7.1.1. Filter . . . 67
7.1.2. Taskfile . . . 68
7.2. Hashketten-basiertes Routing . . . 69
7.2.1. Dauer des Attribute Overlay Aufbaus . . . 69
7.2.2. Suboptimale Pfade . . . 70
7.2.3. Master-Publisher . . . 72
7.3. ACO Routing . . . 74
7.3.1. Dauer des Attribute Overlay Aufbaus . . . 74

8. Ausblick **76**
8.1. Vollständige Asymmetric Dining-Cryptographers Network Implementierung . . . 76
8.2. Routingalgorithmen . . . 76
8.3. Erweiterung des Webbased Graphical User Interface . . . 77
8.4. Evaluationsframework . . . 77

9. Literatur **78**

10. Abkürzungsverzeichnis **81**

11. Abbildungsverzeichnis **83**

12. Tabellenverzeichnis **84**

13. Listingsverzeichnis **85**

Anhang **87**

A. Listings **87**
A.1. Implementierung . . . 87

A.2. Evaluation 89

1 Einleitung

Viele Angebote im Internet, die an Endnutzer gerichtet sind, verfolgen heute einen zentralisierten Ansatz, bei dem die Daten beim Dienstanbieter verarbeitet und gespeichert sind und das Zugriffsgerät des Endnutzers mehr oder weniger nur als Schnittstelle zur Ein- und Ausgabe verwendet wird. Prominente Beispiele für solche zentralisierten Systeme sind soziale Netzwerke wie *Facebook*[1] oder Carsharing-Angebote wie *Uber*[2].

Solche zentralisierten Systeme erlauben dem Dienstanbieter allerdings vielfach den uneingeschränkten Zugriff auf die bei ihm gespeicherten Daten, die er dadurch leicht missbrauchen kann. Brechen Dritte in das System des Anbieters ein oder gibt der Anbieter selbst diese Daten (ungewollt) weiter, so können diese außerdem in die falschen Hände geraten. Prominente Beispiele hierfür lassen sich sowohl bei Facebook [3] als auch bei Uber und anderen Anbietern finden. Zu nennen wäre hier beispielsweise der Datenklau bei Yahoo, bei dem Daten aller drei Milliarden Nutzer abgegriffen wurden [4], oder der Einbruch beim amerikanischen Dienstleister Equifax, dem Daten von fast der Hälfte aller US-Bürger verloren gingen, darunter Sozialversicherungsnummern, Kreditkartendaten, Name, Geburtsdatum und Adressen der Betroffenen, sowie teilweise auch Führerscheinnummern [5]. Im Fall von Uber wurden sogar 100.000 Dollar „Lösegeld“ an die Hacker gezahlt, damit diese die Daten nicht veröffentlichten [6]. Ein Schutz vor solchen Datenlecks ist allerdings nicht inhärent in jedem dezentralen Ansatz enthalten, sondern muss durch weitere Maßnahmen erzeugt werden, die die Anonymität gewährleisten. Auch die Überwachung der Bürger und die Unterdrückung der Meinungsfreiheit, beispielsweise in China, ist mit zentralen Systemen einfacher zu bewerkstelligen, als mit dezentralen.

In der Literatur findet sich ein neuartiger Ansatz zur anonymen Gruppenkommunikation in Peer to Peer (P2P) Netzwerken auf Publish/Subscribe (Pub/Sub)-basis, der auf der Verwendung von *Hashketten* basiert. Dieser wurde allerdings bisher nicht realisiert, sondern nur durch Simulation evaluiert [1]. Darauf aufbauend gibt es einen ebenfalls neuartigen Ansatz, um die Effizienz dieser Gruppenkommunikation mittels *Ant Colony Optimization (ACO)* [2] zu steigern. Ebenfalls gibt es einen Ansatz, die Anonymität der Publisher durch Dining-Cryptographers Network (DC-net)-artige Verschlüsselung noch besser zu schützen [7, 8, 9]. Auch diese beiden Ansätze wurden bisher nur simulativ evaluiert.

Eine simulative Evaluation hat natürlich den Vorteil, dass sie sehr flexibel ist und die zu untersuchenden Konzepte auf einem hohen Abstraktionslevel simuliert werden können, was nicht nur die Evaluation selbst erheblich erleichtert, sondern auch das Verständnis der zu betrachteten Konzepte fördert. Es gibt jedoch auch Abhängigkeiten und Probleme, die in einer simulativen Evaluation nur schwer zutage treten und die erst durch eine prototypische Implementierung und die Umsetzung in einer realen Testumgebung zum Vorschein kommen.
Oftmals wird beispielsweise die hohe Nebenläufigkeit von P2P Netzwerken in Simulationen vereinfacht und serialisiert. Probleme, die aus echter Nebenläufigkeit entstehen können, wie beispielsweise Nachrichten, die sich überholen oder unterschiedliche Laufzeiten aufweisen, werden so vermieden und können deshalb erst in einer prototypischen Implementierung zutage treten. Auch lassen sich in Simulationen Objekte mit Verhalten oder Eigenschaften modellieren, die in der Realität nicht existieren können, wie beispielsweise das Objekt einer Kante in einem Graphen. Im Kontext dieser Masterarbeit ist eine solche Kante eine Netzwerkverbindung zwischen den Teilnehmern des P2P Netzwerks, kann also als solche

[1] siehe `https://www.facebook.com`
[2] siehe `https://www.uber.com`

beispielsweise selbst keine Pheromone o. Ä. speichern. Das können nur die Teilnehmer des P2P Netzwerks selbst.

Diese Masterarbeit schließt diese Lücke mit einer prototypischen Implementierung der genannten Ansätze zur anonymen Gruppenkommunikation, die dadurch auf ihre Realisierbarkeit untersucht werden sollen. Eine solche Implementierung kann darüber hinaus auch gut als Basis für eine spätere Realisierung der Konzepte für Endnutzer verwendet werden und/oder in der Forschung um weitere Anonymisierungstechniken oder Routingalgorithmen ergänzt werden.

Ich werde in Kapitel 2 die Grundlagen behandeln und in Kapitel 3 einige ähnliche und bereits implementierte Systeme vorstellen. In Kapitel 4 werde ich dann die in [1, 10, 2, 7, 8, 9] geschilderten Konzepte erläutern und in den Kontext dieser Masterarbeit setzen. Kapitel 5 wird die Probleme und Lösungen schildern, die die erstmalige prototypische Implementierung dieser Konzepte mit sich geführt hat. Diese Implementierung werde ich dann in Kapitel 6 ausführlich behandeln. Mit einer Evaluation des entstandenen Systems in Kapitel 7 und einem Schlusswort in Kapitel 8 werde ich diese Masterarbeit schließen.

2 Grundlagen

In der vorliegenden Masterarbeit werden einige Begriffe und Konzepte häufig erwähnt, die im Folgenden näher erläutert werden.

2.1 Hashes und Hashketten

Hashfunktionen bilden ein beliebig großes Eingabedatum auf ein kleines Ausgabedatum fester Länge ab, sie sind daher Reduktionsfunktionen, die, anders als Kompressionsalgorithmen, nicht umkehrbar sind. Eine eindeutige Rückrechnung von der Ausgabe einer Hashfunktion $H(x)$, genannt *Hash*, auf die Eingabedaten ist daher nicht möglich. Es gibt viele Hashfunktionen, beispielsweise Cyclic Redundancy Check (CRC) [11] oder auch einfach die Quersumme einer Zahl.

Kryptografische Hashfunktionen haben mehrere zusätzliche Eigenschaften, die beliebige Hashfunktionen nicht haben. Die für diese Masterarbeit wichtigste Eigenschaft ist die *Kollisionsresistenz*. Das bedeutet, dass es – auch wenn theoretisch nicht völlig unmöglich – praktisch unmöglich ist, zwei Eingabedaten zu finden, die den gleichen Hash haben oder zu einem Hash ein passendes Eingabedatum zu berechnen.

2.1.1 Hashketten

Wendet man eine Hashfunktion wiederholt auf ihre eigene Ausgabe an, so ensteht eine Kette von Hashes. Nutzt man eine kryptografische Hashfunktion, so ist es praktisch unmöglich von einem Hash in der Kette auf das Kettenglied davor zu schließen, lediglich die Berechnung der nachfolgenden Kettenglieder ist möglich. Abbildung 2.1 veranschaulicht diesen Sachverhalt: Vom Hash $H_2 := 0xDEAD$ aus gesehen lassen sich die Hashes $H_3 := 0xCAFE$, $H_4 := 0xBABE$ und $H_5 := 0xF00D$ berechnen, nicht jedoch die Hashes $H_1 := 0xBEEF$ oder $H_0 := 0xC0DE$.

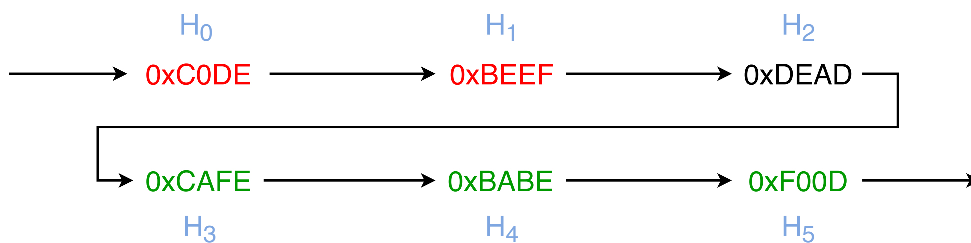


Abbildung 2.1.: Ausschnitt von 6 Hashes einer Hashkette. Die grünen Hashes können von Hash H_2 aus berechnet werden, die roten nicht.

Der Abstand zweier Hashes einer Hashkette lässt sich berechnen, indem beide Hashes so lange immer wieder neu gehasht werden, bis einer der beiden Hashes den ursprünglichen Wert des anderen annimmt. Die Anzahl der benötigten Anwendungen der Hashfunktion ist dann der Abstand der beiden Hashes der Kette. Zwischen H_2 und H_5 beträgt der Abstand beispielsweise drei (H_2 muss dreimal gehasht werden, um H_5 zu erzeugen). Listing 2.1 verdeutlicht diesen Vorgang in Python-Code. Damit die Berechnung des Abstandes zweier Hashes, die *nicht* zur selben Hashkette gehören, nicht endlos läuft, muss die Suche an einer festgelegten Obergrenze für den Abstand abbrechen, welche ich in Einklang mit [10] d_{max} nennen will.

```python
# H1 und H2 sind die gegebenen Hashes, dmax entspricht der gewählten Obergrenze d_max
def caclculate_distance(H1, H2, dmax):
    x1 = H1
    x2 = H2
    count = 0

    while H1 != x2 and H2 != x1 and count < dmax:
        x1 = hash(x1)
        x2 = hash(x2)
        count = count + 1

    return count
```

Listing 2.1: Abstandsberechnung zweier Hashes einer Hashkette

Begriffsdefinition 2.1:
Ist in dieser Arbeit die Rede von Hashes, die *kürzer/kleiner* oder *länger/größer* als andere Hashes sind, so sind die beiden Hashes im Kontext einer Hashkette zu betrachten. Der kürzere Hash ist dabei weiter links in der Hashkette zu finden und kann durch wiederholtes Hashen auf den Wert des längeren Hashes gebracht werden, der weiter rechts in der Hashkette zu finden ist.

2.2 Publish/Subscribe

Das Pub/Sub Paradigma entkoppelt die Erzeuger/Sender von Informationen von den Empfängern dieser Informationen. Dazu müssen die Subscriber, das sind die Empfänger der Informationen, über einen Filter festlegen, für welche Informationen sie sich interessieren und welche sie nicht benötigen. Diese Filterung kann entweder anhand eines Themas, dem *Topic* oder auch *Channel*, geschehen oder auf dem Inhalt der Informationen selbst basieren. Die Veröffentlichung der Informationen kann entweder als steter Informationsstrom (Streaming) oder über diskrete Nachrichten geschehen. In dieser Masterarbeit werden nur Topic-basierte Systeme beschrieben und untersucht, die diskrete Nachrichten verwenden.

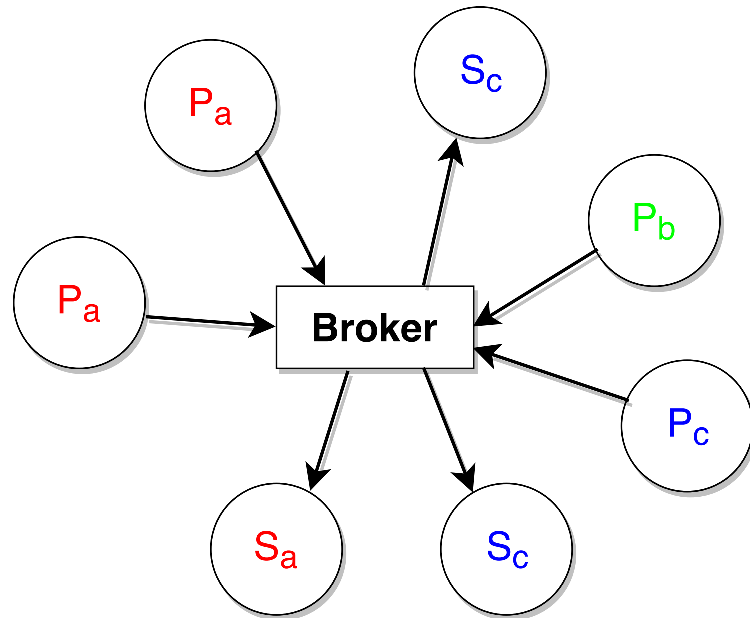
In Pub/Sub Systemen wird die Verteilung der Nachrichten der Publisher an die interessierten Subscriber entweder von einem *Broker* übernommen, der die Nachrichten dann anhand der Topics filtert und weiterleitet, oder die Subscriber verbinden sich direkt mit den Publishern des gewünschten Topics, um von diesen Nachrichten zu erhalten (siehe Abbildung 2.2). In beiden Fällen müssen die Publisher dem Broker oder den Subscribern ein *Advertisement* ihrer Topics schicken, damit diese wissen, welche Topics existieren. Ebenfalls müssen die Subscriber *Subscriptions* an den Broker oder die Publisher schicken, damit diese ihnen dann die Nachrichten schicken können (siehe auch [12]).

Wird ein Broker genutzt, so senden die Publisher ihre Nachrichten alle direkt an den Broker. Die Subscriber wiederum verbinden sich ebenfalls mit dem Broker, welcher dann alle Nachrichten der Publisher, die für ein bestimmtes Topic Informationen publizieren, an die Subscriber weiterleitet, die ein Interesse an diesem Topic angemeldet haben. Dabei weiß weder der Subscriber wie viele Publisher zu einem Topic existieren oder ob es überhaupt Publisher für dieses Topic gibt, noch weiß der Publisher, wie viele Subscriber sich für sein Topic interessieren oder ob sich überhaupt Subscriber für das Topic interessieren.

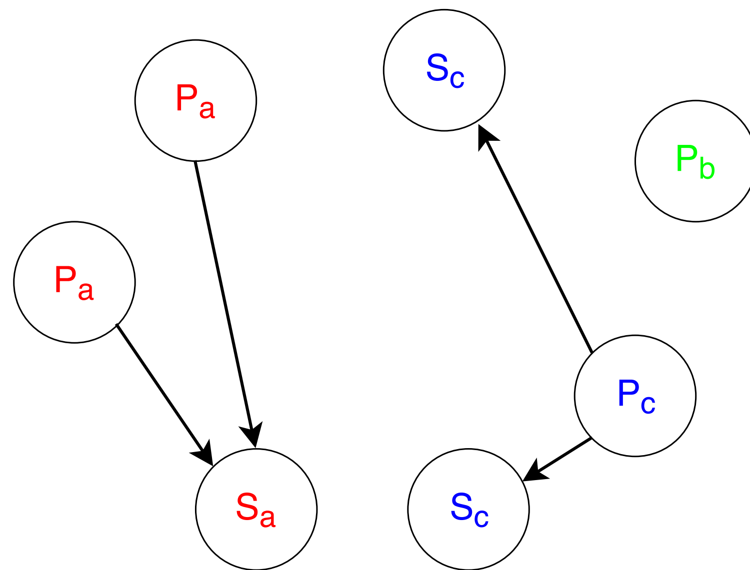
Definition 2.2:
Ein Pub/Sub System M für eine Menge von Topics A besteht aus den Pub/Sub Teilnehmern T mit $M := \{\mathcal{M}_a | \forall a \in A\}$ für ein Topic a aus der Menge aller Topics A und $\mathcal{M} := (\mathcal{P}_a, \mathcal{S}_a)$ mit $\mathcal{P}_a \subseteq T$ und $\mathcal{S}_a \subseteq T$, wobei $\mathcal{P}_a$ die Menge aller Publisher und $\mathcal{S}_a$ die Menge aller Subscriber für das Topic $a \in A$ repräsentieren.

Abbildung 2.2a zeigt die Verteilung der Nachrichten bei Verwendung eines Brokers, der die Nachrichten der beiden Publisher für Topic a (rot) an den Subscriber des Topics weiterleitet. Die Nachrichten des Publishers für Topic b (grün) werden vom Broker nur empfangen und nicht weitergeleitet, da sich kein Subscriber für dieses Topic beim Broker registriert hat. Das Topic c (blau) wird von einem einzelnen Publisher veröffentlicht und von zwei Subscribern empfangen.

Abbildung 2.2b zeigt das gleiche Szenario wie in Abbildung 2.2a, diesmal aber ohne Broker. Folglich verbinden sich die Publisher und Subscriber eines Topics direkt miteinander, sodass alle Publisher mit allen Subscribern dieses Topics verbunden sind, die Subscriber die Nachrichten der Publisher also direkt empfangen.



(a) Pub/Sub mit Broker



(b) Pub/Sub ohne Broker

Abbildung 2.2.: Verschiedene Topologien eines einfachen Pub/Sub Systems

P2P Netzwerke sind dezentrale Netzwerke, in denen jeder Teilnehmer grundsätzlich die gleichen Fähigkeiten und Funktionen hat. Meist setzt ein solches P2P Netzwerk als Overlay auf einem anderen Netzwerk auf, beispielsweise dem Internet. Zwei im P2P Netzwerk benachbarte – also direkt miteinander verbundene – Knoten können dabei im darunterliegenden Netzwerk, genannt Underlay, weit voneinander entfernt und nur über mehrere Zwischenschritte verbunden sein. In dieser Masterarbeit werden nur P2P Netzwerke behandelt, die als Overlay auf einem anderen Netzwerk wie beispielsweise dem Internet aufbauen. Die Begriffe P2P Netzwerk und P2P Overlay sind daher synonym.

Definition 2.3:
Ein P2P Netzwerk lässt sich darstellen als ein Graph $G := (V, E)$ bestehend aus den Knoten V, die über die Kanten $E \subseteq V \times V$ miteinander verbunden sind. Ein P2P Netzwerk ist vollvermascht, wenn gilt: $E := V \times V$.

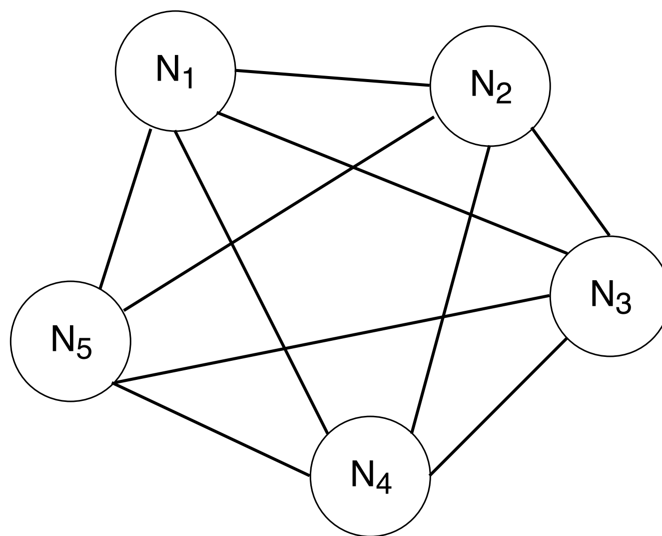


Abbildung 2.3.: Vollvermaschtes P2P Netzwerk mit 5 Knoten

Ist das P2P Netzwerk nicht vollvermascht (für ein Beispiel eines vollvermaschten Netzwerks, siehe Abbildung 2.3), sind also nicht alle Teilnehmer direkt mit allen anderen Teilnehmern verbunden, muss die Kommunikation zwischen zwei Knoten des Netzwerks möglicherweise über Zwischenknoten laufen. Die Auswahl der geeigneten Zwischenknoten, genannt Forwarder, ist die Aufgabe von Routingalgorithmen. Diese lassen sich grob in zwei Klassen einteilen:

global
: Globale Routingalgorithmen haben das Wissen über jeden Knoten und jede Verbindung im Netzwerk und können basierend darauf den optimalen Weg finden, um die Daten zum Ziel zu befördern. Beispiele hierfür sind Link State Routingalgorithmen wie *OSPF* [13] oder *IS-IS* [14].

lokal
: Lokale Routingalgorithmen haben nur die Sicht eines einzelnen Knotens und gegebenenfalls die seiner direkten Nachbarn zur Verfügung und müssen diese limitierte Information nutzen, um die Daten zum Ziel zu bringen. Beispiele hierfür sind Distance Vector Routingalgorithmen wie *RIP* [15]. Ebenfalls lokal sind die Routingalgorithmen *Flooding*, *Gossiping* und *Randomwalk*.

Der *Durchmesser* eines P2P Netzwerks ist der längste aller kürzesten Pfade dieses Netzwerks, lässt sich also durch Gleichung (2.1) berechnen.

$$diameter := max(\forall V_1, V_2 min(distance(V_1, V_2)) \quad (2.1)$$

2.4 Privatsphäre/Anonymität

Daubert et al. definieren Privatsphäre im Kontext von Pub/Sub als Kombination aus Vertraulichkeit und der Anonymität der Teilnehmer, welche beide eng zusammenhängen, da die Vertraulichkeit bei Verletzung der Anonymität kompromittiert werden kann und umgekehrt [1, S. 43, 16]. Für ein genaues Modell eines Angreifers, siehe Abschnitt 2.6.

2.4.1 Anonymität

Folgt man der Definition von Pfitzmann und Hansen [17], so lässt sich die Anonymität quantitativ messen: "Anonymity of a subject (participant) means that the subject is not identifiable within a set of subjects, the anonymity set." [17] Diese Anonymitätsmenge kann in einem beliebigen Kommunikationssystem maximal alle Teilnehmer umfassen, im Kontext von Pub/Sub also maximal alle Knoten des Overlays. Die Anonymität eines Teilnehmers ist daher umso größer, je größer die minimale Anonymitätsmenge ist, in der ein Angreifer ihn lokalisieren kann [1, S. 43].

Ein Angreifer, der die Anonymität von Subscribern oder Publishern brechen will, wird also versuchen die Anonymitätsmenge möglichst auf alle Subscriber oder alle Publisher eines Topics zu reduzieren. Kann ein Angreifer die Größe der Anonymitätsmenge nicht auf weniger als k Teilnehmer reduzieren, wobei k größer sein sollte, als die Anzahl der tatsächlich existierenden Subscriber oder Publisher des betrachteten Topics:

$$\forall a \in A : |anonSet(a)| \geq k \quad (2.2)$$

Ist Gleichung (2.2) erfüllt, so ist die Anonymität von Subscribern oder Publishern im Sinne der oben genannten Definition sichergestellt, da sich die tatsächlichen Subscriber oder Publisher des betrachteten Topics zwischen unbeteiligten Knoten verstecken können [1, S. 43]. Je größer k gewählt werden kann, ohne dass Gleichung (2.2) durch einen Angreifer verletzt werden kann, desto quantitativ höher ist die Anonymität des betrachteten Systems. Für ein genaues Modell eines Angreifers, siehe Abschnitt 2.6.

2.4.2 Vertraulichkeit

Daubert et al. definieren Vertraulichkeit als die geheime/verschlüsselte Übermittlung von Informationen, sodass nur diejenigen Zugriff auf die Informationen haben, für die diese Informationen auch gedacht sind bzw. die diese Informationen benötigen, um die Funktion des betrachteteten Systems zu gewährleisten [1, S. 43]. Im Kontext dieser Masterarbeit bedeutet das daher, dass nur die Subscriber eines Topics a die Nachrichten der Publisher zu Topic a entschlüsseln können und Routinginformationen nur denjenigen Knoten zur Verfügung stehen, die diese zwingend für das Routing benötigen.

2.5 Anonymes Pub/Sub mit P2P Netzwerken

Bringt man nun die beschriebenen Konzepte *Pub/Sub* (Abschnitt 2.2), *P2P Netzwerke* (Abschnitt 2.3) und *Anonymität* (Abschnitt 2.4) zusammen, so kann – bei Verwendung eines geeigneten Routingalgorithmus – ein anonymes Pub/Sub System entstehen.

Globale Routingalgorithmen (beispielsweise basierend auf Steinerbäumen) liefern im Idealfall in Bezug auf die benötigte Bandbreite und die Anzahl der Nachrichtenweiterleitungen optimale Ergebnisse. Solche Routingalgorithmen sind aber im Kontext von anonymer Kommunikation nicht verwendbar, da solche Routingalgorithmen erfordern, dass mindestens eine einzelne Instanz oder aber sogar alle Teilnehmer des P2P Netzwerks detailierte Informationen über den Aufbau des Netzwerks haben. Wird ein solcher Routingalgorithmus verwendet, können Pseudonyme im Netzwerk über das globale Wissen immer einem Teilnehmer des Netzwerks zugeordnet werden. Anonyme Kommunikation ist nicht mehr möglich, da der Weg einer Nachricht durch das globale Wissen allen Knoten lückenlos zur Verfügung steht, siehe dazu auch die Betrachtungen zu Anonymität in Abschnitt 2.4 und das betrachtete Angreifermodell in Abschnitt 2.6.

Klassischerweise werden für das anonyme Routing in P2P Netzwerken die naiven lokalen Routingalgorithmen *Flooding*, *Gossiping* [18] oder *Randomwalk* [19] verwendet:

Flooding

Bei reinem *Flooding* werden dabei die Nachrichten der Publisher über das gesamte P2P Netzwerk geflutet, indem jeder Knoten eine empfangene Nachricht an all seine Nachbarn weiterleitet, sofern er sie nicht schon einmal empfangen hat. Hat er die Nachricht schon einmal empfangen, so wird sie einfach ignoriert, um Schleifen und endlos umherlaufende Nachrichten zu verhindern. Nachrichten können außerdem mit einer Time to Live (TTL) versehen werden, um zu verhindern, dass sie besonders lange durch das Netzwerk irren.

Randomwalk

Bei Verwendung von *Randomwalk* als Routingalgorithmus wird eine oder mehrere einzelne Nachrichten auf einem zufälligen Weg durch das P2P Netzwerk geschickt, sodass diese nach einem genügend langen Lauf bei allen Knoten des Netzwerks vorbeigekommen sind. Dabei wird eine Nachricht bei der Weiterleitung *nicht* dupliziert, wie das bei *Gossiping* der Fall wäre. Oftmals werden die Nachrichten außerdem mit einer TTL versehen, um zu verhindern, dass sie endlos durch das Netzwerk irren und/oder sie werden beim Empfang eines Duplikats, beispielsweise ausgelöst durch eine Schleife, nicht mehr weitergeleitet.

Gossiping

Bei *Gossiping* werden Nachrichten ähnlich wie bei einem Randomwalk über das Netzwerk verteilt. Ein Knoten leidet eine erstmalig empfangene Nachricht allerdings nicht nur an einen einzelnen Nachbarknoten weiter, sondern an eine Untermenge von i Nachbarn. Auch hier verbreitet sich die Nachricht über das ganze Netzwerk und endlos laufende Nachrichten werden wie bei *Randomwalk* durch eine TTL und/oder die Erkennung von Duplikaten verhindert.

Die beschriebenen naiven Routingalgorithmen weisen in der Regel allerdings schlechtere Eigenschaften auf, als ihre globalen Verwandten. Das liegt vor allem daran, dass alle drei Routingalgorithmen die Nachrichten der Publisher an alle Knoten des P2P Netzwerks verbreiten, statt nur an die Subscriber, die sich für die Nachrichten tatsächlich interessieren. Der Overhead in Bezug auf die Anzahl der weitergeleiteten Nachrichten und die verwendete Bandbreite ist daher im Vergleich zu einem globalen Routingalgorithmus,

der die Nachrichten auf dem direkten Weg von den $\mathcal{P}_a$ zu allen $\mathcal{S}_a$ weiterleitet, recht hoch. Bei Gossiping oder Randomwalk kann es außerdem vorkommen, dass eine Nachricht nicht bei allen gewünschten Zielen ankommt, da sie unterwegs verloren gehen kann, beispielsweise weil ein Knoten des Netzwerks während der Weiterleitung das P2P Netzwerk frühzeitig verlässt oder weil die Nachricht eine Schleife läuft oder ihre TTL abgelaufen ist und sie deshalb nicht mehr weitergeleitet wird.

Eine Alternative für die beschriebenen naiven Routingalgorithmen ist daher, für jedes *Topic* (in meiner restlichen Arbeit auch *Channel* oder *Attribut* genannt) einen Verteilbaum aufzubauen, der alle Publisher und Subscriber dieses Topics und zusätzliche Zwischenknoten enthält, die benötigt weden, um alle Publisher und Subscriber dieses Baumes miteinander zu verbinden (siehe auch Abbildung 2.4). Über diesen Verteilbaum können die Publisher dann ihre Datennachrichten an die Subscriber verteilen. Im Kontext von Pub/Sub lassen sich die Zwischenknoten als *Message Broker* auffassen, da sie für die Verteilung der Nachrichten von den Publishern zu den Subscribern sorgen. Im Unterschied zum klassischen zentralen *Message Broker* ist diese Funktionalität hier aber über mehrere Knoten verteilt.

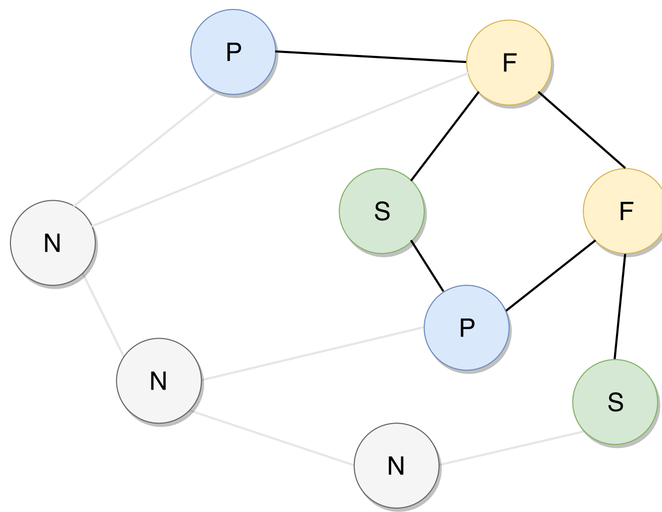


Abbildung 2.4.: Beispiel eines Verteilbaumes. Die grauen Knoten des P2P Netzwerks und deren Kanten sind nicht Teil des Verteilbaumes, die beiden orangenen Forwarder werden dagegen zur Weiterleitung der Nachrichten benötigt, auch wenn sie selbst weder Publisher noch Subscriber sind.

Erweitert man die Definitionen 2.2 und 2.3 um die Beschreibung aus dem letzten Absatz, so ergibt sich die neue Definition 2.4.

Definition 2.4:
Die Menge aller Knoten eines Verteilbaumes $V_a \subseteq V$ ist definiert als $V_a := (\mathcal{M}_a, \mathcal{F}_a) := (\mathcal{P}_a, \mathcal{S}_a, \mathcal{F}_a)$ mit den Forwardern $\mathcal{F}_a$ und den Publishern $\mathcal{P}_a$ sowie den Subscribern $\mathcal{S}_a$. Die Publisher sind die Wurzeln des Verteilbaumes und die Subscriber die Blätter des Baumes.

Der Aufbau eines solchen Verteilbaumes – bei Daubert et al. auch *Attribute Overlay* oder *Attribute Mesh* genannt – lässt sich über *Flooding*, *Gossiping* oder *Randomwalk* erledigen, während das eigentliche Routing von Nachrichten dann nur noch im Verteilbaum stattfindet. Das entlastet das P2P Netzwerk, da nicht jede Nachricht an alle Knoten weitergeleitet wird, erzeugt aber auch einen gewissen Overhead, der durch den Aufbau des Verteilbaumes entsteht. In [1, 10] wird der Aufbau eines Verteilbaumes über *Flooding* beschrieben, wobei Hashketten anstatt einer TTL verwendet werden (zu Hashketten, siehe Abschnitt 2.1).

Ein optimaler Verteilbaum wäre jedoch ein Steinerbaum, der, anders als ein minimaler Spannbaum, auch „unbeteiligte“ Zwischenknoten enthalten kann, sich aber sonst ähnlich zu einem minimalen Spannbaum verhält. In einem solchen Steinerbaum wären die Subscriber optimal mit den Publishern verbunden, das Berechnen eines Steinerbaums benötigt aber globales Wissen über die beteiligten Publisher und Subscriber sowie den Aufbau des P2P Netzwerks. In [2] nähern Grube et al. deshalb den Steinerbaum mittels ACO an, ohne dabei globales Wissen einzusetzen. In [20, 21] zeigt Gutjahr, dass die optimale Lösung mittels ACO beliebig gut angenähert werden kann.

2.6 Angreifermodell

Ein Angreifer auf die Anonymität des in Abschnitt 2.5 geschilderten Systems kann mit verschiedenen Eigenschaften beschrieben werden:

passiv oder aktiv

Ein *passiver* Angreifer kann das P2P Netzwerk nur beobachten und die ausgetauschten Nachrichten belauschen, wohingegen ein *aktiver* Angreifer in das P2P Netzwerk eingreifen und Nachrichten fälschen/verändern, umleiten oder löschen kann.

intern oder extern

Ein *interner* Angreifer kontrolliert einen oder mehrere Knoten des P2P Netzwerks und hat daher Zugriff auf Verschlüsselungsschlüssel und ähnliche „geheime“ Informationen, die nur den Teilnehmern des Netzwerks zur Verfügung stehen. Im Kontrast dazu besitzt ein *externer* Angreifer kein solches internes Wissen.

lokal oder global

Der Angreifer kann entweder *globales* Wissen über den gesamten Aufbau des P2P Netzwerks besitzen, wozu auch gehört, dass er die Kommunikation aller Knoten untereinander abhören kann, oder er kann als *lokaler* Angreifer nur die Kommunikation eines oder einiger weniger Knoten abhören.

Die von Daubert et al. vorgeschlagenen Anonymisierungstechniken (siehe Abschnitt 4.6) sollen gegen einen globalen Angreifer schützen, der als passiver externer Beobachter den Netzwerkverkehr jedes einzelnen Knotens verfolgen kann. Er besitzt also die Information über $G := (V, E)$ und versucht $\mathcal{M}$ oder die Untermengen $\mathcal{P}_a$ oder $\mathcal{S}_a$ für ein von ihm gewähltes Attribut $a \in A$ zu finden. Um das zu erreichen kann er statistische Analysen und Timing Analysen anwenden, durch die er herausfinden kann, welche Knoten die Empfänger einer publizierten Nachricht, also die Subscriber, sind. Diese Knoten bleiben auch bei Restrukturierungen des P2P Netzwerks und des *Attribute Overlays* (siehe Abschnitt 2.5) konstant die Ziele der Nachrichten, leiten eine Nachricht also nicht mehr an andere Knoten weiter und sind daher durch die Timing Analyse als Blatt des Verteilbaumes erkennbar.

3 Related Work

Neben dem hier vorgestellten anonymen Pub/Sub System auf P2P-basis gibt es auch noch andere Systeme zur anonymen Kommunikation, die auch schon implementiert sind und eine breite Nutzerbasis haben:

The Onion Router (Tor)[3]

Tor leitet den Datenverkehr durch mehrere Knoten des Netzwerks, bevor er dann über einen Exitnode in das Internet entlassen wird [22]. Dadurch ist nicht mehr erkennbar, von welcher IP-Adresse aus die Anfrage ursprünglich stammt. Nur noch die IP-Adresse des Exitnodes ist sichtbar.

Damit der Pfad einer Anfrage nicht nachverfolgt werden kann, wendet Tor Onion Routing an. Dabei wird der Datenverkehr zwiebelartig mehrfach verschlüsselt. An jedem Knoten des Tor Netzwerks wird dann eine Verschlüsselungsschicht entschlüsselt und die Anfrage dann weiter zum nächsten Knoten geschickt. Erst der Exitnode sieht die Anfrage dann im Klartext. Dieser Knoten kann aber nicht mehr feststellen, von wo die Anfrage ursprünglich kam. Knoten auf dem Pfad der Anfrage kennen nur den jeweils vorhergehenden und nachfolgenden Knoten, kennen durch die Verschlüsselung aber weder den Inhalt der Anfrage, noch das Ziel im Internet, an die die Anfrage gehen soll. Auch den Absender der Anfrage kennen sie (bis auf den ersten Knoten) nicht.

The Invisible Internet Project (I2P)[4]

I2P ist ein P2P-basiertes Netzwerk, das – ähnlich zu Tors Onion Routing – Garlic Routing mit mehreren Verschlüsselungsschichten verwendet [23]. I2P bietet als Overlay über dem Internet – ähnlich zum IP-Protokoll [24, 25] – paketvermittelte Dienste an, die als Adressen kryptografische Schlüssel verwenden.

Da I2P somit die gleiche Basis bietet, wie das Internet selbst, lassen sich im I2P Overlay alle Dienste anbieten und benutzen, die auch im klassischen Internet angeboten werden können. Dazu gehören Webseiten, Maildienste, Chatdienste oder auch Blogsysteme. Alle diese Dienste sind durch die Verwendung von kryptografischen Schlüsseln zur Adressierung und das Garlic Routing völlig anonym nutzbar.

Eine Verbindung zum klassischen Internet, die, ähnlich wie in Tor, über Exitnodes zur Verfügung gestellt wird, ist zwar technisch möglich, wird aber nicht als Ziel von I2P gesehen.

Freenet-Project[5]

Freenet arbeitet, anders als Tor oder I2P, als anonymer verteilter Speicher [26, 27]. Die einzelnen Knoten des P2P Netzwerks werden hier verwendet, um einen Teil der Daten des Netzwerks zu speichern. Das eingesetzte Routing stellt sicher, dass die Daten von anderen Knoten im P2P Netzwerk gefunden und anonym abgerufen werden können. Dazu werden die Daten unter einem eindeutigen kryptografischen Identifier mehrfach, also redundant, im Netzwerk abgelegt. Inhalte, die oft abgerufen werden, werden dabei von vielen Knoten lokal vorgehalten und können daher auch von allen Teilnehmern schnell geladen werden. Daten, die selten abgerufen werden, sind dagegen auf nur wenigen Knoten gespeichert und können daher auch nicht von allen Teilnehmern zuverlässig abgerufen werden. Nicht mehr benötigte und daher nicht mehr abgerufene Inhalte werden mit der

3 Zu finden unter `https://www.torproject.org/`
4 Zu finden unter `https://geti2p.net/`
5 Zu finden unter `https://freenetproject.org/`

Zeit auf immer weniger Knoten des Systems gespeichert und verlassen das P2P System irgendwann endgültig.

Die Auslegung als verteilter Speicher macht Freenet im Vergleich zu Tor oder I2P größtenteils statisch. Es existieren zwar Möglichkeiten veröffentlichte Daten zu aktualisieren, aber eine solche Aktualisierung kann einige Minuten dauern. In Extremfällen kann es sogar sein, dass das Update verloren geht und/oder nicht von allen Knoten abrufbar ist.

Die beschriebenen Systeme sind entweder zu statisch (Freenet) oder bieten nur eine $1{:}n$-Kommunikation, aber keine $n{:}m$-Kommunikation, wie sie bei Pub/Sub Systemen möglich ist. Gerade im Bereich von Internet of Things (IoT) Geräten ist aber eine effiziente $n{:}m$-Kommunikation wichtig.

Freenet ermöglicht zwar eine $n{:}m$-Kommunikation, da n Teilnehmer Daten im verteilten Speicher ablegen können, die dann m Teilnehmer parallel abrufen können, allerdings ist Freenet trotzdem größtenteils ein statischer verteilter Speicher. Zwar gibt es eine Möglichkeit Daten zu versionieren, sodass eine gewisse Dynamik möglich ist, allerdings ist Freenet durch die Auslegung als verteilter Speicher eher träge und für Echtzeitkommunikation oder nahezu Echtzeitkommunikation nicht geeignet. Gerade im Bereich IoT, bei dem viele Geräte in nahezu Echtzeit untereinander und mit dem Nutzer kommunizieren sollen, ist die Trägheit von Freenet zu groß.

Möchte man stattdessen allerdings Tor oder I2P für anonyme $n{:}m$-Kommunikation verwenden, so verlangt das Design dieser Systeme, dass zwischen jedem der Kommunikationsteilnehmer eine eigene Verbindung mit eigenen Verschlüsselungsschichten aufgebaut werden muss. Dabei gibt es keine Möglichkeit gemeinsame Pfade zwischen mehreren Teilnehmern effizient zu nutzen, so wie das Multicast beispielsweise im klassischen Internet tut [28]. Dadurch entsteht ein signifikanter *Message Overhead* [2].

Pub/Sub Systeme erlauben eine $n{:}m$-Kommunikation. Werden sie anonym gestaltet, so können sie eine in Bezug auf *Message Overhead* effizientere Kommunikation ermöglichen, als Tor oder I2P dazu in der Lage sind. Die Untersuchung und Entwicklung von anonymen Pub/Sub Systemen auf P2P Basis soll daher eine effizientere $n{:}m$-Kommunikation ermöglichen, als das mit bereits etablierten Systemen machbar ist.

4 Konzepte

Die für den Aufbau eines anonymen P2P basierten Pub/Sub Netzwerks benötigten Konzepte sollen in diesem Kapitel erläutert werden. Jeweils ein Abschnitt wird sich allgemeinen Konzepten wie der Attributlokalisierung oder den Management Messages, den Optimierungszielen, dem Flooding- und Hashketten-basierten Routing von Daubert et al. sowie dem ACO Routing von Grube et al. widmen. Zuletzt werden dann die von Daubert et al. vorgeschlagenen Anonymisierungstechniken vorgestellt.

4.1 Allgemeine Konzepte

Um die in Abschnitt 2.4.2 genannte Vertraulichkeit zu gewährleisten, müssen die ausgetauschten Informationen so verschlüsselt sein, dass nur die Teilnehmer des P2P Netzwerks Zugriff darauf haben, die diese Informationen auch benötigen und berechtigt sind auf diese zuzugreifen, sodass ein lokaler passiver oder auch aktiver Angreifer diese nicht mitlesen oder manipulieren kann. Die hierfür verwendeten Schlüssel müssen den jeweiligen Teilnehmern des Netzwerks verfügbar gemacht werden. Daubert et al. schlagen dazu die Verwendung einer Trusted Third Party (TTP) vor, die als zentrale Instanz diese Informationen verwaltet und an die Teilnehmer des P2P Netzwerkes verteilt [1, S. 45].

Eine solche TTP ist aber ein Single Point of Failue (SPoF) und als solcher vermutlich das Hauptangriffsziel eines Angreifers, der mit der Kenntnis der von der TTP ausgegebenen Schlüssel, die durch diese hergestellte Vertaulichkeit im kompletten P2P Netzwerk und damit auch die Anonymität aller Teilnehmer brechen kann. Daubert et al. sind sich dessen bewusst und haben die TTP daher als *offline* konzipiert, sodass die Schlüsselvergabe von der Benutzung des P2P Netzwerks entkoppelt ist [1]. Ob die Verwendung einer TTP sich in der Praxis so umsetzen lässt und welche Vor- oder auch Nachteile aus der Verwendung einer TTP entstehen, ist allerdings nicht Gegenstand dieser Masterarbeit und wird daher weder im Prototyp verwendet, noch hier weiter behandelt.

Gegenstand dieser Masterarbeit ist hingegen die Untersuchung, wie verschiedene Routingalgorithmen und Anonymisierungstechniken verhindern oder zumindest erschweren können, dass die (statistische) Analyse von Metadaten oder Verkehrsdaten, die trotz Verschlüsselung anfallen und von Angreifern erhoben werden können, zum Brechen der Anonymität der Teilnehmer des P2P Netzwerks verwendet werden können.

4.1.1 Lokalisierung von Publishern und Subscribern eines Topics (Attributlokalisierung)

Damit die Subscriber die Daten der Publisher empfangen können, müssen beide Parteien eine Verbindung über das P2P Netzwerk aufbauen. Der Aufbau einer solchen Verbindung kann konzeptionell entweder geschehen, indem die Publisher alle Knoten des P2P Netzwerks darüber informieren, dass sie Daten zu einem oder mehreren bestimmten Topics veröffentlichen, sodass sich interessierte Subscriber daraufhin melden können, oder indem die Subscriber das Netzwerk gezielt nach Publishern eines Topics durchsuchen [12].

Eine zentrale Stelle, die den Subscribern eine Liste der Topics und ihrer Publisher zur Verfügung stellen würde (oder umgekehrt den Publishern eine Liste der Subscriber und Topics für die sich diese interessieren), würde die Anonymität aushebeln (siehe auch Abschnitt 2.4) und wird in dieser Arbeit daher nicht weiter betrachtet.

4.1.2 Management Messages

Daubert et al. schlagen die Nutzung von UDP [29] vor, damit sichergestellt ist, dass keine unerwarteten Information Leaks auftreten [1, S. 46]. Da UDP im Gegensatz zu TCP verbindungslos ist, schlagen Daubert et al. vor, dass beide Verbindungspartner Heartbeat Messages verschicken, um zu erkennen, ob eine Verbindung noch intakt ist. Diese Heartbeat Messages sind periodisch versendete Nachrichten, die jeder Knoten mit seinen direkten Nachbarknoten austauscht. Empfängt ein Knoten über einen bestimmten Zeitraum keine einzige Heartbeat Message, so wird die Verbindung zum Nachbarknoten als abgebrochen betrachtet und diese Verbindung aus den Routingtabellen und/oder sonstigen Strukturen entfernt. Solche Heartbeat Messages können beispielsweise alle 250 Millisekunden versendet und die Verbindung als abgebrochen betrachtet werden, wenn vier Sekunden lang keine solche Heartbeat Message empfangen wurde.

Alle Management Messages, die für den Aufbau und die Instandhaltung der Attribute Overlays benötigt werden, wie beispielsweise die Ameisen von ACO oder die gefluteten Advertisements des Hashketten-basierten Routings von Daubert et al., werden in diesen Heartbeat Messages versteckt. Diese werden außerdem mittels Padding immer auf eine fest definierte Größe gebracht, sodass der globale passive Angreifer die im Heartbeat versteckten Management Messages nicht anhand ihrer Größe erkennen kann [1, S. 46]. Lediglich reine Datennachrichten werden außerhalb der Heartbeat Messages übertragen, damit diese nicht durch die Größe und die Frequenz der Heartbeat Messages begrenzt werden. Wie alle Nachrichten zwischen direkten Nachbarknoten, werden auch die Heartbeat Nachrichten verschlüsselt übertragen.

Damit Paketverluste, die die Heartbeat Messages betreffen, nicht zum Verlust wichtiger Management Messages führen, werden Management Messages durchnummeriert und die höchste empfangene Nummer vom empfangenden Knoten an den Absender zurückgemeldet. Sind eine oder mehrere Management Messages bis zum Senden der nächsten Heartbeat Message noch nicht als empfangen bestätigt, so werden sie erneut versendet. Das sorgt dafür, dass auch bei Paketverlust alle Management Messages ohne Verlust durch das P2P Netzwerk geleitet werden. Hohe Paketverluste können die Management Messages allerdings verzögern. Wie stark sich solche Verzögerungen in der Praxis auswirken, werde ich in Kapitel 7 näher untersuchen.

Ist die Round Trip Time (RTT) größer als das Intervall der Heartbeat Messages, so werden die in der Heartbeat Message versteckten Management Messages auch ohne Paketverlust mehrfach verschickt. In Gleichung (4.1) wird der Zusammenhang zwischen der Anzahl der nötigen Übertragungen einer Management Message T_{mm} und der RTT des zugrunde liegenden Netzwerks t_{rtt} (in Sekunden) sowie dem Intervall der Heartbeat Messages t_{hb} (ebenfalls in Sekunden) verdeutlicht. Die theoretisch maximal für Management Messages verfügbare Datenübertragungsrate R_{mm} (in Bytes pro Sekunde) unter Betrachtung der Größe einer Heartbeat Message S_{hb} (in Bytes) und der Anzahl der nötigen Übertragungen einer Management Message T_{mm} zeigt Gleichung (4.2). Diese theoretisch maximale Datenübertragungsrate begrenzt die Anzahl der Management Messages, die pro Sekunde zwischen zwei Knoten ausgetauscht werden können. Übersteigt die für ausgetauschte Management Messages benötigte Datenübertragungsrate dauerhaft den zur Verfügung stehenden Wert, so stauen sich die Management Nachrichten im betroffenen Knoten an und das P2P Netzwerk kann Funktionsprobleme aufweisen. Overhead bei Management Messages ist daher möglichst zu vermeiden.

$$T_{mm} := ceil(\frac{t_{rtt}}{t_{hb}}) \tag{4.1}$$

$$R_{mm} := \frac{S_{hb}}{t_{hb} * T_{mm}} \tag{4.2}$$

Diese theoretisch maximale Datenübertragungsrate hängt, wie die beiden Gleichungen zeigen, auch von der RTT des dem P2P Overlay zugrunde liegenden Netzwerks ab. Je nach Implementierung kann die tatsächliche Datenübertragungsrate außerdem geringer ausfallen als hier berechnet, da das verwendete Verfahren zum Packen der Management Messages in die Heartbeat Messages durch das verwendete Encoding und/oder andere Implementierungsdetails einen Overhead erzeugen können, der die Datenübertragungsrate weiter begrenzt.

Beispiel: Bei einer RTT von 100 Millisekunden (t_{rtt}), einem Heartbeat Intervall von 250 Millisekunden (t_{hb}) und einer Größe von 1200 Bytes pro Heartbeat Message (S_{hb}) wird eine theoretisch mögliche maximale Datenübertragungsrate von 4800 Bytes pro Sekunde erreicht. Bleiben alle Werte Parameter gleich, aber die RTT erhöht sich auf 300 Millisekunden (t_{rtt}), so beträgt die theoretisch mögliche maximale Datenübertragungsrate nur noch 2400 Bytes pro Sekunde, da sich T_{mm} von Eins auf Zwei erhöht.

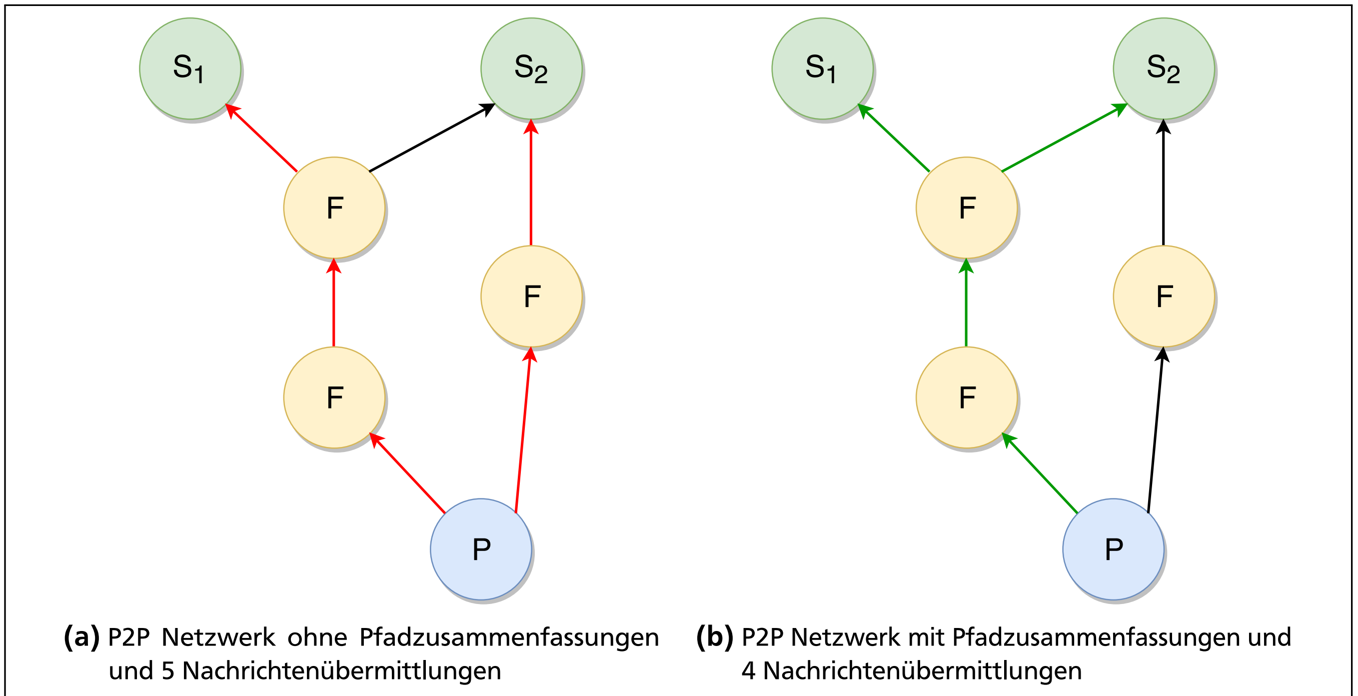
4.2 Optimierungsziel

Ziel der Routingalgorithmen von Daubert et al. und insbesondere ACO von Grube et al. ist es, den Attribute Overlay so aufzubauen, dass die Übermittlung der Datennachrichten im P2P Netzwerk bezüglich des dafür notwendigen Overheads möglichst effizient ist. Der Begriff *Overhead* bezieht sich hierbei auf den Vergleich von direkter Kommunikation ohne P2P Overlay mit der Kommunikation über ein P2P Netzwerk. Je geringer der Overhead gehalten wird, desto kleiner ist die Verzögerung, die durch die Verwendung eines P2P Netzwerks zwangsläufig entsteht, da jede Weiterleitung im Netzwerk extra Zeit kostet.

Je stärker sich die Pfade zwischen einem Publisher und seinen verschiedenen Subscribern zu einem gemeinsamen Anfangsabschnitt vereinen, während gleichzeitig die Summe aller Pfadlängen möglichst gering bleibt, desto weniger Datennachrichten müssen übertragen werden, wodurch der Overhead verringert wird.

Abbildung 4.1 zeigt ein Beispiel einer solchen Optimierung mit Pfadzusammenfassungen. In Abbildung 4.1a werden zwei getrennte Pfade zu den Subscribern S_1 und S_2 verwendet, die zusammen fünf Nachrichtenübermittlungen zur Folge haben. In Abbildung 4.1b sind dagegen die beiden Pfade im Anfangsabschnitt zusammengefasst. Auch wenn in Abbildung 4.1b der Pfad zu S_2 länger ist als vorher, so werden durch die Zusammenfassung trotzdem nur noch vier Nachrichtenübermittlungen insgesamt benötigt, der Overhead ist hier also geringer als vorher.

Ebenfalls optimierungwürdig ist die Zahl der ausgetauschten Management Messages, denn je mehr Management Messages ausgetauscht werden müssen, bis der Attribute Overlay aufgebaut ist, desto größer ist die Verzögerung, die entsteht. Mit einer steigenden Zahl von Management Messages wird außerdem auch eine höhere Datenübertragungsrate benötigt um mit dieser Zahl Schritt zu halten. Die an einem Knoten verfügbare Datenübertragungsrate ist aber eine begrenzte Ressource, die möglichst wenig beansprucht werden sollte.



(a) P2P Netzwerk ohne Pfadzusammenfassungen und 5 Nachrichtenübermittlungen

(b) P2P Netzwerk mit Pfadzusammenfassungen und 4 Nachrichtenübermittlungen

Abbildung 4.1.: Optimierung eines P2P Netzwerks durch Pfadzusammenfassungen

4.3 Hashketten-basiertes Routing

In dem von Daubert et al. vorgeschlagenen Hashketten-basierten Routing fluten die Publisher den P2P Overlay mit der Information, dass sie auf einem bestimmten Channel publishen (d.h. ein bestimmtes Topic publishen, siehe auch Abschnitt 4.1.1). Interessieren sich Subscriber für einen Channel, dann aktivieren sie im P2P Overlay einen Pfad zum jeweiligen Publisher.

Dafür definieren Daubert et al. in ihrem Routingalgorithmus eine *Advertisement Phase* und eine *Subscription Phase* [10, S. 688f, 1, S. 46f], ihr vorgeschlagener Algorithmus ist daher nach [12] Advertisement-basiert.

Daubert et al. nutzen eine TTP, um Schlüssel für die verschiedenen Schichten der Verschlüsselung im System zu verteilen. Der hier betrachtete Prototyp arbeitet allerdings ohne Verschlüsselung, eine TTP ist also weder implementiert, noch ist deren Einbindung oder die Zuordnung der verschiedenen Schlüssel zu den ausgetauschten Informationen in dieser Arbeit näher erklärt. Für nähere Informationen zur Verwendung einer TTP, siehe [10, 1].

4.3.1 Advertisement Phase

In der *Advertisement Phase* fluten die Publisher den P2P Overlay mit Advertisements für das von ihnen publizierte Topic. Da die Knoten des P2P Overlays in der Lage sein müssen, verschiedene Publisher für dasselbe Topic voneinander unterscheiden zu können und Duplikate sowie Schleifen beim Fluten erkannt werden sollen, benötigt jeder Publisher eine eindeutige Identifikation und es muss eine TTL verwendet werden. Das bringt laut Daubert et al. aber Probleme bei der Anonymität mit sich:

> *To prevent overlay partitioning, forwarders must be able to distinguish duplicates from the same publisher and two different publishers $p_1, p_2 \in P_a$. However, publisher IDs contradict anonymity. Random time-to-live counters would allow adversaries to lie. This would enable them to capture more messages and to partition the overlay. [10, S. 688]*

Daubert et al. schlagen deshalb vor, für jedes Advertisement einen *Hash* als Transaktionspseudonym zu verwenden. Ein Transaktionspseudonym ist ein zufällig gewähltes Pseudonym, welches nur für eine einzelne Transaktion, in diesem Fall das Fluten des Advertisements über das ganze P2P Netzwerk, gilt und nicht mehrfach von einem Knoten verwendet wird. Als solches ist ein Transaktionspseudonym daher nicht auf den dazugehörigen Knoten, also dessen eindeutige ID oder IP-Adresse etc., zurückzuführen und verletzt die Anonymität dieses Knotens daher nicht. Verschiedene Transaktionen eines einzelnen Knotens lassen sich auch nicht über ein gemeinsames Pseudonym zusammenführen, da jede Transaktion ein neues und zufällig gewähltes Transaktionspseudonym verwendet.

Begriffsdefinition 4.1:
Ein Advertisement, welches den Hash h_1 enthält, ist dann *kleiner* oder *kürzer* als ein anderes Advertisement mit dem Hash h_2, wenn h_1 kleiner als h_2 ist, also h_1 weiter links in der Kette steht als h_2 und durch wiederholtes Hashen auf den Wert von h_2 gebracht werden kann. Ein Advertisement ist hingegen *größer* oder *länger*, wenn h_1 größer als h_2 ist (siehe auch Abschnitt 2.1.1).

Jeder Knoten des P2P Overlays, der ein Advertisement weiterleitet, hasht den empfangenen Hash erneut und schickt diesen anstelle des empfangenen Hashs im ausgehenden Advertisement weiter, wodurch eine Hashkette gebildet wird. Ob zwei Hashes zur selben Kette gehören und damit das gleiche Transaktionspseudonym repräsentieren, kann von jedem Knoten eigenständig berechnet werden. Für diese Berechnung muss nur einer der Hashes so lange immer wieder gehasht werden, bis der Wert des zweiten Hashes entsteht (siehe auch Abschnitt 2.1.1 und [10, S. 688]).
Damit eine solche Suche nicht endlos läuft, wenn zwei Hashes nicht zur selben Kette gehören, muss außerdem eine Obergrenze für den maximalen Abstand zweier Hashes definiert werden.

Jeder Knoten speichert den Hash, das Topic und den Nachbarknoten, über den der Hash empfangen wurde in seiner Routingtabelle. Gespeichert wird dabei immer nur der bis jetzt kürzeste Hash und der dazugehörige Empfangsknoten nebst Topic. Ist bereits ein größerer Hash gespeichert, wird dieser Eintrag durch den neuen Eintrag ersetzt und das Advertisement in diesem Fall auch nicht weiter geflutet, da es schon einmal geflutet wurde (Duplikatserkennung) [10, S. 688].

Bei Verwendung einer TTL kann ein aktiver interner Angreifer, der einen Knoten kontrolliert, diese dekrementieren und dadurch andere Knoten glauben machen, dass der kürzeste Pfad zu einem Publisher über den Knoten des Angreifers führt. Dadurch bekommt der interne Angreifer mehr Datenverkehr zu sehen, den er dann untersuchen und/oder manipulieren kann. Die Verwendung von kryptografischen Hashketten sorgt dagegen dafür, dass es einem aktiven internen Angreifer nicht möglich ist, einen kleineren Hash der Kette zu berechnen (sprich: die „TTL“ zu dekrementieren), um damit mehr Datenverkehr auf sich umzuleiten. Das erschwert es einem aktiven internen Angreifer an mehr Datenverkehr zur Analyse bzw. Manipulation zu kommen.

Ungenauigkeit: Zusammenhang zwischen Obergrenze d_{max} und Netzwerkdurchmesser

In [10] schreiben Daubert et al., d_{max} müsse gemäß dem erwarteten maximalen Durchmesser des Netzwerks gewählt werden, erklären aber nicht den genauen Zusammenhang zwischen dem Durchmesser und d_{max}.

Tatsächlich muss d_{max} so gewählt werden, dass ein Advertisement, welches einen Knoten auf verschiedenen Wegen erreicht, immer noch als Dopplung erkannt werden kann. Der Abstand der Hashes zueinander hängt dabei von der Topologie des P2P Netzwerks ab, steht damit also auch lose im Zusammenhang zum Durchmesser dieses Netzwerks. d_{max} muss mindestens so groß gewählt werden wie der im Netzwerk maximal entstehende Abstand zweier Hashes der gleichen Kette, damit durch nicht erkannte Dopplungen keine Probleme im Routing entstehen. Solche Probleme können beispielsweise Overhead beim Fluten durch nicht erkannte Dopplungen oder auch doppelt abgesendete Subscriptions, also mehrfach aktivierte Datenpfade, sein.

Umgekehrt kann der maximale Durchmesser des P2P Overlays deshalb *mindestens* so groß sein, wie die gewählte Obergrenze d_{max}. Bei geschickter Positionierung des Publishers und/oder der Forwarder im Overlay kann der erlaubte Durchmesser des Overlays sogar wesentlich höher sein. Abbildung 4.2 macht diesen Sachverhalt grafisch deutlich. Hierbei werden Laufzeitunterschiede im P2P Underlay angenommen, die dazu führen, dass die gefluteten Advertisements so wie gezeigt ankommen. Alternativ kann aber auch einfach angenommen werden, dass die Knoten mit dem jeweils nächstgrößeren Hash das Netzwerk zeitlich nacheinander betreten. Damit die Subscriber in Abbildungen 4.2a bis 4.2d und der Publisher in Abbildung 4.2e die empfangenen Hashes als zur selben Hashkette zugehörig erkennen können, muss d_{max} jeweils mindestens den in der Abbildung angegebenen Wert haben. Dieser entspricht jedoch nicht dem Durchmesser der gezeigten Netzwerke.

Allgemein lässt sich sagen: Speichern die Publisher den Zufallswert, der dem gesendeten Hash zugrunde liegt, in ihrer Routingtabelle, als hätten sie diesen als Hash empfangen, dann ist der durch d_{max} vorgegebene Mindestdurchmesser, der sicher möglich ist, durch Gleichung (4.3) gegeben. Speicherten die Publisher jedoch den Hash aus dem gesendeten Advertisement in ihrer Routingtabelle, würde für den Mindestdurchmesser $floor(\frac{d_{max}}{2})$ gelten.

$$durchmesser_{max} = ceil(\frac{d_{max}}{2}) \tag{4.3}$$

Ungenauigkeit: Äquivalenz von TTL und Obergrenze d_{max}

Daubert et al. schreiben, dass die beschriebene Obergrenze (bei ihnen d_{max} genannt) äquivalent zu einer TTL wäre [10, S. 688]. Da Daubert et al. nicht erklären, was sie unter einer TTL verstehen, gehe ich in der folgenden Betrachtung daher von der Definition als *Hop Limit* aus, die auch in [25] gebraucht wird. Die TTL ist demnach ein Datenfeld in jeder Nachricht, welches mit einem initialen positiven Wert gefüllt wird (in unserem Fall mit d_{max}) und dann bei jeder Weiterleitung im P2P Netzwerk um Eins verringert wird. Ist der Wert bei Null angekommen, so ist die „Lebenszeit“ der Nachricht vorüber und die Nachricht wird aus dem Netzwerk entfernt und nicht mehr weitergeleitet. Vergleiche ich nun den Weg einer Nachricht im P2P Netzwerk bei der Verwendung einer TTL mit dem Anfangswert d_{max} mit dem Weg dieser Nachricht bei Verwendung einer Hashkette, so werden deutliche Unterschiede sichtbar. Abbildung 4.3 zeigt diese

(a) Beispielnetzwerk mit Durchmesser 3 und minimal notwendigem d_{max} von 0

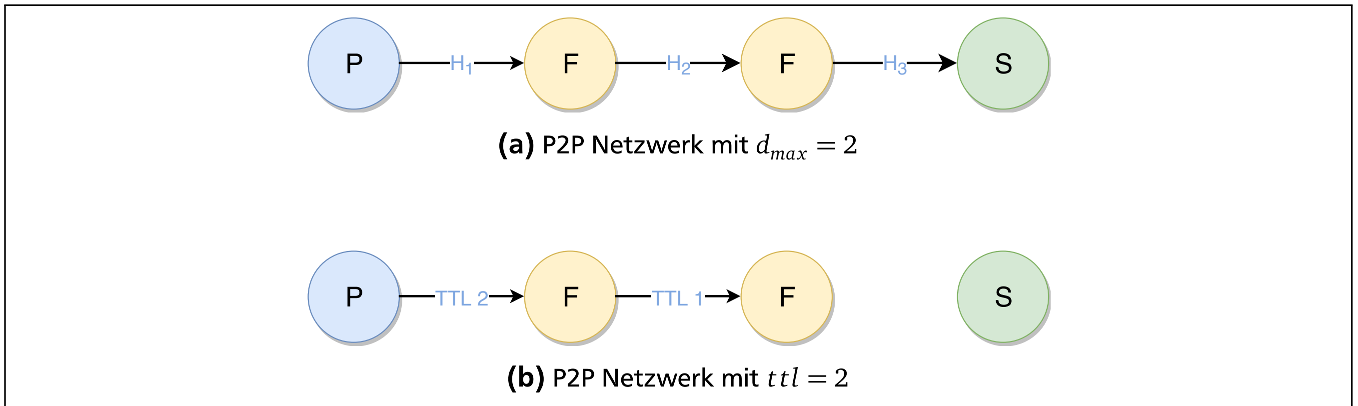
(b) Beispielnetzwerk mit Durchmesser 6 und minimal notwendigem d_{max} von 0

(c) Beispielnetzwerk mit Durchmesser 5 und minimal notwendigem d_{max} von 2

(d) Beispielnetzwerk mit Durchmesser 3 und minimal notwendigem d_{max} von 5

(e) Beispielnetzwerk mit Durchmesser 5 und minimal notwendigem d_{max} von 7

Abbildung 4.2.: Relation von Netzwerkdurchmesser und Obergrenze d_{max} bei verschiedenen Topologien



(a) P2P Netzwerk mit $d_{max} = 2$

(b) P2P Netzwerk mit $ttl = 2$

Abbildung 4.3.: Vergleich von TTL als *Hop Count* und d_{max}

Unterschiede grafisch. Wie schon Abbildung 4.2a verdeutlicht, gilt für d_{max} hier $0 \leq d_{max} < \infty$, wählt man beispielsweise $d_{max} = 2$, so funktioniert das Routing in Abbildung 4.3a problemlos. Eine TTL von zwei sorgt aber im gleichen Fall für ein nicht funktionierendes Routing, wie Abbildung 4.3b zeigt.

Definiert man die TTL hingegen als eine echte Zeitangabe (beispielsweise gemessen in Sekunden), so lässt sich die Obergrenze bei der Abstandsberechnung zweier Hashes einer Kette (d_{max}), welche keinen Begriff einer Zeit kennt, *nicht* mit einer TTL vergleichen, die den Begriff der Zeit kennt.

Beide hier beschriebenen Definitionen einer TTL sind deshalb *nicht* äquivalent zu der Obergrenze d_{max}, die Daubert et al. in [10] definieren, weshalb die Behauptung einer Äquivalenz unter diesen Gesichtspunkten zumindest so lange nicht haltbar ist, wie Daubert et al. keine Definition einer TTL angeben, für die eine solche Äquivalenz gezeigt werden kann.

4.3.2 Subscription Phase

Empfängt ein Subscriber ein Advertisement für ein Topic, für das er sich interessiert, so schickt er eine Subscription an den Publisher, die den empfangenen Hash enthält und im P2P Overlay den aktuell bekannten kürzesten Pfad zum Publisher nimmt. Dazu schaut jeder Knoten entlang des Weges in seiner Routingtabelle nach, zu welchem Eintrag, also zu welcher Hashkette, der empfangene Hash gehört und zu welchem nächsten Knoten die Subscription deshalb weitergeleitet werden muss.

Jeder Knoten, den die Subscription passiert, merkt sich dabei den Eingangs- und Ausgangsknoten, sodass ein *Attribute Overlay* für das jeweilige Topic entsteht. Publisher können in diesem Attribute Overlay dann einfach ihre zu veröffentlichenden Daten verteilen.

4.4 ACO Routing

Das in [2] vorgestellte Verfahren nimmt das Verhalten echter Ameisen als Vorbild, um mittels ACO das Optimierungsproblem der Erzeugung eines Attribute Overlays zu lösen.

Echte Ameisen laufen auf der Futtersuche zuerst einmal willkürlich umher (Randomwalk). Waren sie bei der Suche erfolgreich, so markieren sie den gelaufenen Weg bei der Rückkehr zum Ameisenhügel mit Pheromonen. Andere Ameisen auf Futtersuche lassen sich von diesen Pheromonen am Boden leiten,

sodass sie tendenziell eher dem Weg folgen, den eine Ameise durch ihre Pheromonausschüttung bereits markiert hat. Mit der Zeit werden immer mehr Ameisen den markierten Weg gegangen sein, sodass dieser Weg immer stärker mit Pheromonen markiert ist. Aus einem „Feldweg“ wird eine „Landstraße“ und dann eine „Autobahn“ (siehe [2, S. 2]).

ACO orientiert sich nun an dem Vorbild der echten Ameisen, wobei diese die Funktion einer Subscription Message übernehmen. Gutjahr zeigt in [20, 21], dass ACO zu einer optimalen Lösung konvergiert, wenn die beiden Parameter *Ameisenzahl* und *Verdampfungsfaktor* günstig gewählt wurden. Der Verdampfungsfaktor sorgt dafür, dass alte Pheromone langsam verdampfen (also verschwinden), sodass die Lösung nicht gegen lokale Optima konvergieren kann. Eine höhere Zahl der gleichzeitig verwendeten Ameisen erhöht die Geschwindigkeit, mit der die Lösung konvergiert.

Grube et al. teilen den Prozess der Erzeugung eines Attribute Overlays mittels ACO in drei Phasen ein, die in dieser Reihenfolge mehrfach hintereinander durchlaufen werden [2, S. 3]. Das mehrfache Losschicken von Ameisengruppen sorgt dafür, dass sich die von den Ameisen erzeugte Lösung immer weiter dem gesuchten Optimum annähern kann. Würden nur ein einziges Mal Ameisen losgeschickt, so würden die Ameisen dieser Gruppe sich untereinander kaum durch Pheromone beeinflussen und es würde keine optimale Lösung entstehen.

Alle drei Phasen zusammen ergeben eine *Ameisenrunde* und sind in den folgenden Abschnitten näher beschrieben.

4.4.1 Phase 1: Prepare Round

In dieser Phase erzeugen die Subscriber eine Anzahl von Ameisen, die sie später auf die Suche schicken wollen. Die Anzahl der gleichzeitig losgeschickten Ameisen und der gewählte Verdampfungsfaktor für die Pheromone beeinflussen, wie gut und wie schnell die Optimierung der Ameisen gegen die optimale Lösung konvergiert [20, 21].

Jede Ameise besitzt einen *Strictness Parameter* σ, der durch eine zufällige Zahl im Bereich $[0, 20)$ repräsentiert wird [2, S. 3]. Dieser Parameter gibt also an, wie stark sich eine Ameise an den Pheromonen orientiert, die auf einer Kante liegen. Eine für jede einzelne Ameise zufällige Wahl dieses Parameters ist wichtig, da nur so eine Balance gefunden werden kann zwischen Ameisen, die die Pheromone auf einem bereits gefundenen guten Pfad erneuern und Ameisen, die explorativ neue Wege suchen. Das Erneuern der Pheromone ist wichtig, damit diese nicht vollständig verdampfen, solange noch kein besserer Pfad gefunden wurde, während das Suchen neuer Wege wichtig ist, um Lösungen zu finden, die eventuell besser sind als die bereits gefundenen.

Jede Ameise besitzt außerdem ein TTL-Feld, welches verhindert, dass Ameisen zu lange bzw. suboptimale Pfade laufen bzw. finden. Die TTL wird mit $min(6, round)$ initialisiert und bei jeder Weiterleitung um eins verringert. Durch die anfangs kleine TTL, die dann schrittweise erhöht wird, werden die Ameisen gezwungen, kürzere und damit optimalere Pfade zu finden, während gleichzeitig auch weit entfernte Publisher irgendwann gefunden werden können, sobald die TTL groß genug geworden ist.

In dieser Phase werden in jeder Runde auch die Pheromone verdampft, indem die Pheromone jeder Kante mit dem Verdampfungsfaktor γ multipliziert werden, wobei dieser Faktor aus dem Intervall $[0, 1]$ gewählt wird und für das gesamte P2P Netzwerk konstant und einheitlich bleibt. Die Verdampfung der Pheromone sorgt dafür, dass sich Lösungen, die nicht optimal sind, nicht im Netzwerk einbrennen.

4.4.2 Phase 2: Searching

In dieser Phase laufen die Ameisen zufällig durch den P2P Overlay (Randomwalk), wobei sie bei der Wahl des nächsten Knotens von den Pheromonen beeinflusst werden, sodass sie nicht völlig zufällig einen Nachbarknoten wählen. Die Pheromone auf einer Kante beeinflussen die Ameisen dahingehend, dass sie tendenziell eher entlang einer Kante mit vielen Pheromonen zum dahinterliegenden Nachbarknoten laufen. Dadurch konvergieren ähnliche Pfade mit jeder Ameisenrunde weiter zu einem gemeinsamen Pfad. Teilen sich mehrere Subscriber einen gemeinsamen Pfad, so ist das besser als die Verwendung zweier getrennter Pfade (siehe Abschnitt 4.2).

Der Strictness Parameter σ gibt nun an, wie stark sich die Ameisen auf ihrem Weg von den Pheromonen auf den Kanten beeinflussen lassen. Je höher dieser Parameter ist, desto höher ist die Wahrscheinlichkeit, dass eine Ameise auf ihrem zufälligen Weg einer Kante mit Pheromonen folgt. Grube et al. geben mit Gleichung (4.4) die Wahrscheinlichkeit c_k an, mit der ein Nachbarknoten n_k für den nächsten Schritt ausgewählt wird [2, S. 3]. $e_{n_k}.\tau$ repräsentiert dabei die Pheromone auf der Kante zum Nachbarknoten n_k und $a_j.\sigma$ steht für den Strictness Parameter der betrachteten Ameise j.

$$c_k = \frac{1}{|Neighbors|} + \frac{(e_{n_k}.\tau)^{a_j.\sigma}}{\Sigma_{n_m \in Neighbors}(e_{n_m})^{a_j.\sigma}} \tag{4.4}$$

Rückfragen bei den Autoren ergab jedoch, dass die Gleichung (4.4) einen Fehler enthält. Die eigentliche Intention der Autoren war es, die Wahrscheinlichkeit für die Wahl der nächsten Kante zur Hälfte zufällig und zur Hälfte abhängig von den Pheromonen auf den Kanten zu machen. Gleichung (4.5) ist eine korrigierte Fassung der ursprünglichen Gleichung aus [2], die diese Intention nun korrekt widerspiegelt.

$$c_k = \frac{0,5}{|Neighbors|} + \frac{(e_{n_k}.\tau)^{a_j.\sigma}}{2 * \Sigma_{n_m \in Neighbors}(e_{n_m})^{a_j.\sigma}} \tag{4.5}$$

Erreicht die TTL einer Ameise den Wert Null, so wird diese Ameise aus dem System entfernt. Stößt eine Ameise auf einen Publisher für das Topic, für das sie auf der Suche ist, so wird sie in die Liste der zurückkehrenden Ameisen eingefügt, die dann in Phase 3 zu ihrem Subscriber zurückkehren und dabei Pheromone ausschütten (siehe auch Abschnitt 4.4.3).

Damit die Ameisen keine Schleifen bilden, besitzen sie eine Liste der bereits besuchten Knoten. Diese Liste wird auch benutzt, um in Phase 3 den Weg zurückzufinden. Entdeckt eine Ameise eine Schleife, so wird sie ebenfalls aus dem System entfernt.

4.4.3 Phase 3: Returning

In dieser Phase laufen alle Ameisen, die nicht durch die Schleifenerkennung oder eine abgelaufene TTL aus dem System entfernt wurden, zurück zum Subscriber, der sie losgeschickt hat, und schütten dabei Pheromone an den besuchten Kanten aus. Diese Pheromone beeinflussen dann die Wegwahl der Ameisen in der Suchphase späterer Runden. In dieser Phase wird *keine* TTL mehr auf die Ameisen angewendet, da diese einfach den vorher aufgezeichneten Pfad zurücklaufen.

Grube et al. schlagen in [2] vor, die ausgeschütteten Pheromone dabei für jeden Subscriber, zu dem die Ameise gehört, getrennt zu speichern. Das soll dafür genutzt werden können, die Pheromone umgekehrt proportional zu bereits von Ameisen des gleichen Subscribers ausgeschütteten Pheromonen auf einer Kante auszuschütten. Grube et al. geben für das Verhältnis an Pheromonen die Gleichung (4.6) an, wobei $e.\tau_{s_i}$ die bereits ausgeschütteten Pheromone der Ameisen des Subscribers s_i auf der gerade betrachteten Kante e repräsentieren.

$$ratio = \frac{e.\tau_{s_i}}{e.\tau} \tag{4.6}$$

Grube et al. schlagen außerdem vor, jeder Ameise nur ein limitiertes Budget an Pheromonen zur Verfügung zu stellen, die sie auf die besuchten Kanten verteilen kann [2, S. 4].

Nach Rückfragen bei den Autoren wurde jedoch klar, dass neuere Versionen der Verwendung von ACO zur Erzeugung eines Attribute Overlays ohne die in [2] beschriebene Budgetierung der Pheromone funktionieren und auch die in [2] noch beschriebene umgekehrt proportionale Ausschüttung von Pheromonen nicht verwendet wird. Allerdings soll in zukünftiger Forschung eventuell ein System entwickelt werden, bei dem verschiedene Ameisen unterschiedliche Mengen an Pheromonen ausschütten.

4.5 Aktive Pfade bei ACO und Hashketten-basiertem Routing

Damit sowohl beim Hashketten-basierten Routing als auch bei ACO die publizierten Informationen von einem Publisher zu den Subscribern laufen können, müssen diese Pfade aktiviert und damit dem Attribute Overlay hinzugefügt werden. Die aktiven Pfade repräsentieren daher einzelne Pfade innerhalb des Attribute Overlays. Mehrere aktive Pfade können sich dabei Abschnitte des Weges teilen, beispielweise nach der in Abschnitt 4.2 erklärten Zusammenfassung durch Pfadoptimierungen.

Bei Daubert et al. wird das Konzept der aktiven Pfade kurz angeschnitten, wenn erklärt wird, wie die Subscription Routingtabellen gefüllt werden [10, S. 689]. Es wird allerdings nicht im Detail geschildert, wie diese Pfade auf- und abgebaut oder instand gehalten werden sollen. Die Funktionsweise aktiver Pfade will ich deshalb hier kurz erläutern.

Aktive Pfade müssen an den Knoten mit folgenden Informationen gespeichert werden:

Routinginformationen (Routing von Datennachrichten)
Damit das eigentliche Routing entlang des aktiven Pfades funktioniert, müssen sowohl der eingehende Nachbarknoten eines aktiven Pfades, als auch der ausgehende Nachbarknoten gespeichert werden. Es können durch Pfadzusammenfassungen o. Ä. auch mehrere aktive Pfade mit dem gleichen eingehenden und ausgehenden Nachbarknoten gespeichert sein.

Quelle und Ziel des Pfades (Pfadidentifikation)
Damit verschiedene Pfade unterschieden werden können, muss für jeden Pfad die Quelle und das Ziel gespeichert werden. Zur Speicherung der Quelle kann statt der IP des Publishers oder einer Knoten-ID auch ein für jedes Topic wechselndes Pseudonym verwendet werden, um die Anonymität zu wahren. Gleiches gilt für die Speicherung des Ziels (der Subscriber).

Die Informationen, um welchen Publisher und welchen Subscriber es sich handelt, wird benötigt, um im Fehlerfall den Pfad abzubauen und/oder den Subscriber über den Fehler auf dem Pfad zu

informieren, sodass dieser einen neuen Pfad aufbauen kann. Außerdem wird die Information über den Subscriber benötigt, um den Pfad bei einem Unsubscribe vollständig abbauen zu können. Die Information über den Publisher hingegen ist ebenfalls nötig, um im Fall eines Unpublish den Pfad ebenfalls abbauen zu können.

Versionierung
Die einzelnen aktiven Pfade müssen versioniert sein, damit ein neu aufgebauter Pfad vom selben Publisher zum selben Subscriber dafür sorgt, dass alte Pfade oder Teile eines alten Pfades zwischen diesem Publisher und Subscriber korrekt abgebaut werden können. Für ACO ist das besonders wichtig, da die Ameisen beim Konvergieren auf die optimale Lösung hin die aktiven Pfade mehrfach ändern können.

Fällt ein Forwarder auf einem aktiven Pfad aus, muss der Nachbarknoten, der auf dem aktiven Pfad in Richtung des Subscribers liegt, diesen Ausfall bemerken und den Subscriber durch eine *Error* Message darüber informieren. Dieser kann dann einen neuen aktiven Pfad aufbauen. Der Nachbarknoten auf dem aktiven Pfad, der in Richtung des Publishers liegt, sendet dagegen eine *Teardown* Message zum Publisher, die dafür sorgt, dass der aktive Pfad für diese Kombination aus Publisher und Subscriber abgebaut wird. Die Versionierung sorgt dabei dafür, dass ein zwischenzeitlich neu entstandener Pfad zwischen Publisher und Subscriber nicht wieder abgebaut wird, sondern nur der alte defekte Pfad.

Der aktive Pfad vom Subscriber bis zum ausgefallenen Forwarder wird von der Error Message *nicht* abgebaut. Möchte man keine alten „herumhängenden“ Teile von aufgegebenen aktiven Pfaden haben, so kann dieser Teil ebenfalls durch eine Teardown Message abgebaut werden, sobald der neue aktive Pfad vom alten Pfad abweicht.

Wird ein solcher Pfadabbau nicht gewünscht, so hat das keinen direkten Nachteil. Ein Ausfall eines weiteren Forwarders auf diesem aufgegebenen aktiven Pfad sorgt aber dafür, dass eine neue Error Message in Richtung des Subscribers verschickt wird. Diese Error Message wird nicht mehr weitergeleitet, wenn ein Forwarder auf dem Pfad der Error Message bereits einen neuen aktiven Pfad für den betroffenen Subscriber kennt. Erreicht die Error Message allerdings den Subscriber, so sorgt sie dafür, dass dieser erneut versucht einen aktiven Pfad aufzubauen, sollte er keinen haben.

Möchte ein Subscriber keine Informationen mehr empfangen, so sendet er eine Unsubscribe Message entlang seiner aktiven Pfade zu allen Publishern, wobei dieser Pfad dabei abgebaut wird. Die aktiven Pfade anderer Subscriber werden dabei nicht verändert, auch wenn sich diese überschneiden sollten. Möchte hingegen ein Publisher nichts mehr publishen, sendet er eine Unpublish Message entlang der aktiven Pfade zu allen Subscribern dieses Publishers. Die aktiven Pfade werden dabei ebenfalls abgebaut.

4.6 Anonymisierungstechniken

Daubert et al. [1] sowie Grube [9] schlagen verschiedene Anonymisierungstechniken vor, um das System gegen einen passiven globalen Angreifer zu härten.

Die von Daubert et al. vorgeschlagenen Verfahren *Probabilistic Forwarding* und *Shell Game* versuchen beide die Anonymität der Subscriber, die von Grube vorgeschlagenen *Covergroups* auf Basis eines Asymmetric Dining-Cryptographers Network (ADC-net) dagegen die Anonymität der Publisher gegen einen solchen Angreifer zu sichern.

4.6.1 Probabilistic Forwarding

Probabilistic Forwarding (PF) versucht das Anonymity Set zu vergrößern, indem dem Attribute Overlay weitere Knoten hinzugefügt werden. Dazu entscheidet jeder Subscriber beim Senden einer Subscription zufällig, welche seiner Nachbarknoten aus dem P2P Overlay er zum Attribute Overlay hinzufügen will [1, S. 48]. Das Hinzufügen solcher *Cover Nodes* findet daher beim Erzeugen oder Erweitern des Attribute Overlays um weitere Subscriber statt. Das bedeutet, dass die Forwarder dem Attribute Overlay zeitgleich mit dem Subscriber beitreten, den sie schützen sollen. Durch diese Gleichzeitigkeit ist es dem externen passiven Angreifer nicht möglich, die neu hinzugekommenen Knoten nach Forwarder oder Subscriber zu unterscheiden.

Solche Cover Nodes können ihrerseits auch wieder weitere Nachbarknoten aus dem P2P Overlay zum Attribute Overlay hinzufügen, sodass ganz neue Äste im Verteilbaum des Attribute Overlays entstehen, die ohne PF nicht da wären. Abbildung 4.4a zeigt den ursprünglichen Verteilbaum mit einem Publisher und vier Subscribern, Abbildung 4.4b den gleichen Verteilbaum nach dem Hinzufügen einzelner Cover Nodes, Abbildung 4.4c zeigt diesen Verteilbaum nach dem Hinzufügen weiterer Cover Nodes, die wiederum Nachbarknoten der ersten Cover Nodes sind. S_1 und S_3 werden in diesem Beispiel durch PF, geschützt, da sie für den Angreifer wie Forwarder aussehen, während F_2 und F_3 (später dann F_4) für ihn aussehen wie Subscriber.

Alle hinzugefügten Knoten werden ein vollwertiger Teil des Attribute Overlays und können daher von einem globalen Angreifer nicht von „normalen“ Knoten des Attribute Overlays unterschieden werden [1, S. 48]. Das bedeutet außerdem, dass solche *Cover Nodes*, die am Ende der Weiterleitungskette stehen, für einen passiven globalen Angreifer aussehen wie Subscriber, da sie die Blätter des Verteilbaumes darstellen. Die echten Subscriber sehen für diesen Angreifer dagegen wie unbeteiligte Forwarder aus.

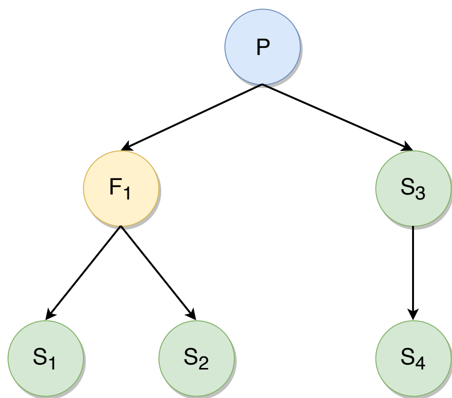
Da PF neue Knoten zum Attribute Overlay hinzufügt, wird der Overhead des Attribute Overlays verschlechtert. Es bekommen nun Knoten Datennachrichten, die diese eigentlich nicht benötigen, da sie weder Subscriber des Topics sind, zu dessen Attribute Overlay sie hinzugefügt wurden, noch Forwarder, die benötigt werden, um echte Subscriber mit dem Publisher zu verbinden.

Darüber hinaus ändert PF ebenfalls nicht die innere Topologie des Attribute Overlays, sodass der globale passive Angreifer diese Information immer noch für die statistische Analyse verwenden kann[1, S. 48]. Das liegt darin begründet, dass die Erweiterung des Attribute Overlays durch die Laufzeit der dafür notwendigen Management Messages zeitlich leicht verzögert zur Erzeugung der „echten“ aktiven Pfade stattfindet. Ein Angreifer kann diese Komposition daher eventuell beobachten und daraus ableiten, was das „Innere“ des Attribute Overlays ist (hier sind die echten Subscriber zu finden) und was das „Äußere“ ist (hier sind nur Cover Nodes zu finden).

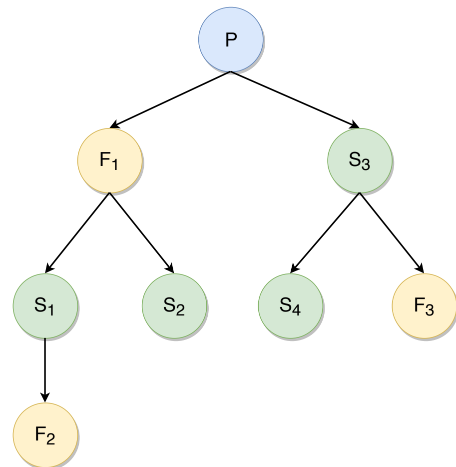
PF hat ebenfalls den Nachteil, dass die Auswahl der zusätzlichen Forwarder nur beim Aufbau des Attribute Overlays oder beim Hinzukommen eines neuen Subscribers durchgeführt wird. Verlassen einige dieser zusätzlichen Forwarder das Netzwerk, so werden sie nicht durch neue ersetzt und die durch sie „versteckten“ Subscriber können dadurch wieder für den globalen passiven Angreifer erkennbar werden.

4.6.2 Shell Game

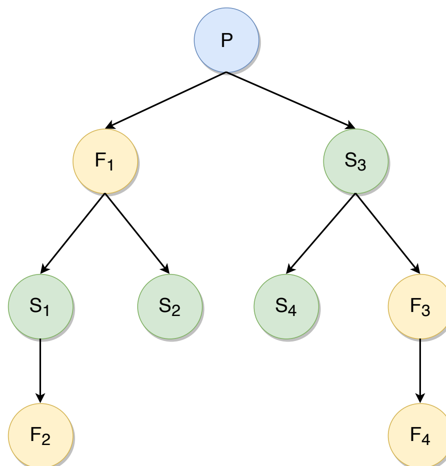
Das Shell Game (SG) soll, wie auch PF, verschleiern, welche Knoten tatsächlich Subscriber und welche nur Forwarder sind. Dazu wird direkt nach dem Aufbau des Attribute Overlays die Topologie dieses Overlays



(a) Ursprünglicher Verteilbaum



(b) Verteilbaum nach einmaliger Anwendung von PF



(c) Verteilbaum nach mehrfacher Anwendung von PF

Abbildung 4.4.: Verschiedene Stadien des Verteilbaumes bei Anwendung von PF

durchgemischt, noch bevor es zum Verteilen von Datennachrichten verwendet wird. Hierbei werden keine neuen Knoten zum neuen Overlay hinzugefügt, wie das bei PF der Fall ist, wodurch sich der Overhead bei der Datenübermittlung auch nicht vergrößert. Durch das neu Mischen der bestehenden Knoten wird aber, ähnlich zu PF, dafür gesorgt, dass Subscriber keine Blätter des Verteilbaums mehr sind. Der Overhead an Management Messages, die zum Durchführen des SG nötig sind, ist jedoch höher, als das bei PF der Fall ist.

Dazu tauschen zwei benachbarte Knoten des Attribute Overlays ihre kompletten Netzwerkverbindungen des P2P Netzwerks aus und nehmen so den Platz des jeweils anderen im Overlay ein. Hierfür werden die vollständigen Routingtabellen zwischen den beiden Knoten ausgetauscht und mittels eines *Two-Phase-Commit* Protokolls wird sichergestellt, dass ein Knoten nicht mit mehreren anderen Knoten gleichzeitig den Platz tauscht.

Ein Knoten beginnt mit dem SG sobald der erste Subscriber verbunden ist [1, S. 48], wobei ein Knoten mehrfach hintereinander den Platz mit einem zufällig gewählten anderen Knoten tauschen kann. Die Wahrscheinlichkeit eines Platztausches nimmt dabei mit der Zeit ab, bis sie schließlich Null erreicht und die Topologie des Overlays damit gefestigt ist. Abbildung 4.5 zeigt zwei Schritte von SG. Der Knoten mit einem roten Rand ist der aktive Knoten, der den Platz mit seinem Nachbarn tauschen wird, getauschte Knoten sind mit gestricheltem Rand kenntlich gemacht.

Da ein Knoten selbstständig entscheiden kann, ob er das SG durchführen möchte, ist es für einen aktiven internen Angreifer möglich, große Teile oder sogar den ganzen Attribute Overlay zu traversieren, indem er wiederholt die Position mit anderen Knoten in diesem Attribute Overlay wechselt, bis er seine Position mit allen Knoten des Attribute Overlays getauscht hat. Dadurch erhält er ein präzises Bild des Attribute Overlays. Da das System dezentral ist, können die anderen Knoten nicht erkennen, dass eine solche Traversierung stattfindet.

Die Übermittlungen von Datennachrichten im Attribute Overlay darf erst beginnen, wenn das SG abgeschlossen ist, damit der globale passive Angreifer nicht den Aufbau des Overlays ohne SG beobachten kann. Durch die Dezentralität ist es aber nicht trivial, den Publisher eines Attribute Overlays zu informieren, sobald alle Knoten ihr SG abgeschlossen haben. Auch wenn mir dieses Problem prinzipiell lösbar scheint, so ist es doch nicht trivial. In einer Simulation tritt dieses Problem dagegen nicht auf, da eine Simulation immer zentral ausgewertet wird und so leicht ermittelt werden kann, wann das SG abgeschlossen ist.

Daubert et al. sprechen an einer Stelle im Paper davon, dass die Knoten bei der Ausführung eines Schrittes des SG alle ihre Verbindungen zu anderen Knoten des P2P Netzwerks tauschen [1, S. 47]. Etwas später sprechen Daubert et al. dagegen davon, dass die Knoten lediglich ${N_a}^-$ und ${N_a}^+$ austauschen, also die Verbindungen, die zum Attribute Overlay gehören, für das gerade das SG durchgeführt wird [1, S. 48]. Das bedeutet jedoch, dass durch das SG zwar nicht die Struktur des Attribute Overlays, dafür aber die Struktur des P2P Netzwerks verändert wird. Eine solche Strukturänderung ist durch den globalen passiven Angreifer beobachtbar und kann genutzt werden um Rückschlüsse auf die ursprüngliche Struktur des Attribute Overlays zu treffen.

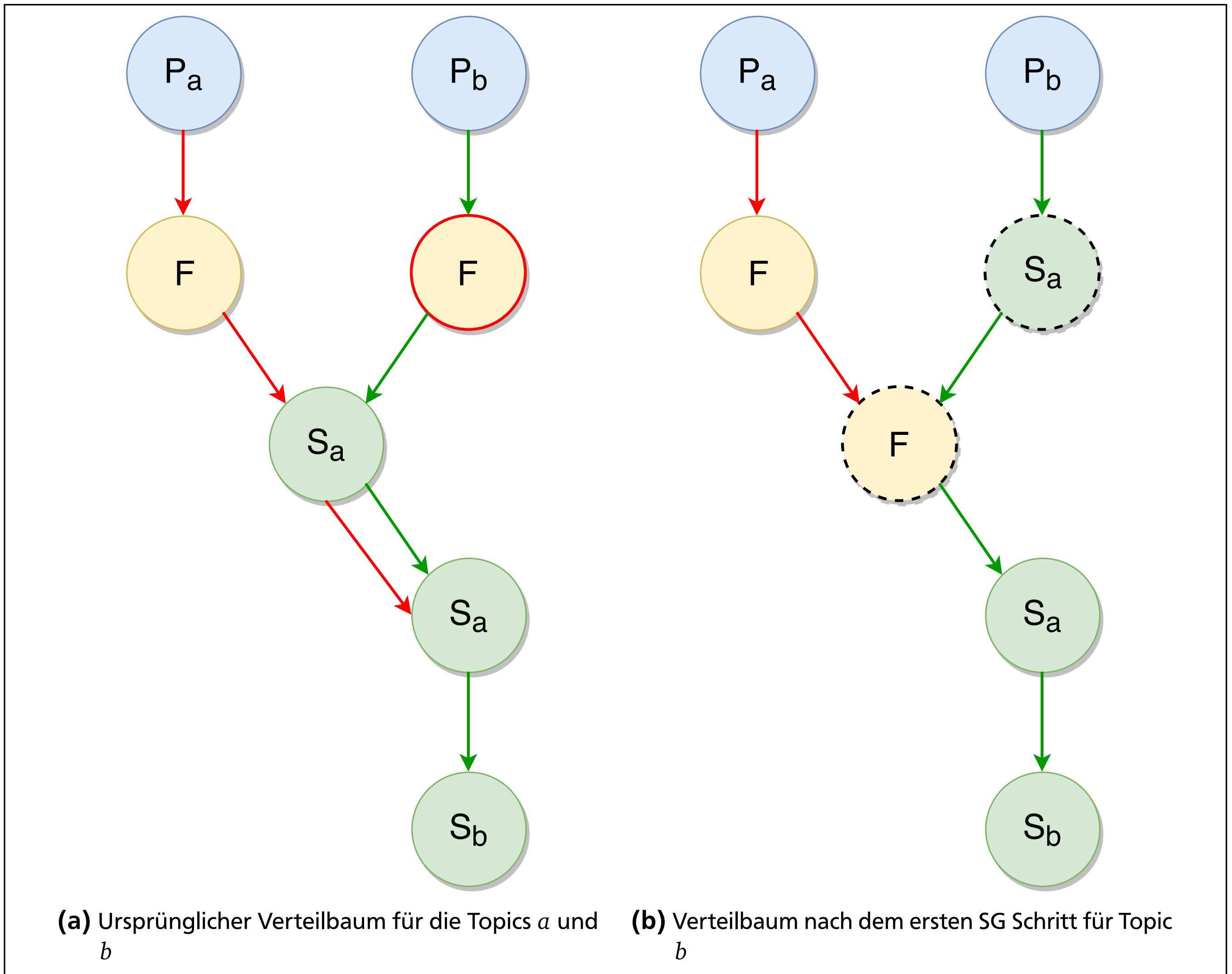
Ein globaler passiver Angreifer kann zwar nicht die Management Messages beobachten, die für das SG benötigt werden; da durch das SG aber neue Netzwerkverbindungen zwischen Knoten aufgebaut und andere Verbindungen dafür abgebaut werden, kann er diese Informationen nutzen, um zu ermitteln, welche Knoten ihre Positionen getauscht haben und damit den ursprünglichen Attribute Overlay rekonstruieren.

(a) Ursprünglicher Verteilbaum

(b) Verteilbaum nach dem ersten SG Schritt

(c) Verteilbaum nach dem zweiten SG Schritt

Abbildung 4.5.: Verschiedene Stadien des Verteilbaumes bei Anwendung von SG



(a) Ursprünglicher Verteilbaum für die Topics a und b

(b) Verteilbaum nach dem ersten SG Schritt für Topic b

Abbildung 4.6.: Destruktion des Attribute Overlays für a durch das SG im Overlay für b

Werden nicht nur die Verbindungen ausgetauscht, die zum Attribute Overlay gehören, für das gerade ein SG stattfindet, sondern alle Verbindungen der Knoten, kann es dazu kommen, dass durch das SG für Topic b ein bereits aufgebautes Attribute Overlay von Topic a zerrissen wird, sodass dort die Datennachrichten nicht mehr ihr Ziel erreichen können. Abbildung 4.6 verdeutlicht diesen Vorgang. In Abbildung 4.6a ist der Attribute Overlay für Topic a mit zwei Subscribern und einem Publisher bereits aufgebaut und in Betrieb (rote Pfeile), der Attribute Overlay für Topic b ist zwar ebenfalls schon aufgebaut (grüne Pfeile), führt aber noch einen Schritt im SG durch, bevor es für die Übermittlung von Datennachrichten genutzt wird. In diesem Schritt wird der rot umrandete Forwarder seine Position mit dem benachbarten Subscriber tauschen. In Abbildung 4.6b sieht man, dass dadurch der Attribute Overlay von Topic b intakt bleibt (alle Subscriber dieses Topics sind weiterhin mit grünen Pfeilen verbunden), der Attribute Overlay von Topic a dagegen unterbrochen wurde und *beide* Subscriber für Topic a keine Datennachrichten mehr erhalten können (die roten Pfeile reichen nicht bis zu den Subscribern).

Nimmt man all die angesprochenen Probleme zusammen, so erscheint mir das Konzept SG nicht prototypisch umsetzbar. Ich habe es folglich auch in dieser Masterarbeit nicht weiter behandelt oder implementiert.

4.6.3 Covergroups (Asymmetric Dining-Cryptographers Network)

Mit dieser Technik soll verschleiert werden, welche Knoten tatsächlich Publisher sind. Dazu bilden mehrere Knoten des P2P Overlays eine Gruppe, die mindestens einen echten Publisher enthält. Die Knoten einer Gruppe müssen dafür im P2P Overlay *nicht* benachbart sein. Ist die Gruppe gebildet, so senden alle Knoten der Gruppe im gleichen Intervall Daten an einen Rendezvous Subject (RS), der stellvertretend für den oder die Publisher der Gruppe als Publisher im P2P Overlay auftritt und die Daten im entsprechenden Attribute Overlay veröffentlicht.

DC-net wurde zuerst von Chaum in [7] vorgeschlagen. Hierbei tauschen alle Teilnehmer mit jedem anderen Teilnehmer einen Schlüssel aus. Jeder Teilnehmer verschlüsselt dann seine Daten mittels XOR mit allen Schlüsseln, die er ausgetauscht hat, bevor er das Ergebnis veröffentlicht. In [8] wird dieses Schema dann unter dem Namen *ADC-net* erweitert, sodass nicht mehr jeder Teilnehmer die Schlüssel aller anderen Teilnehmer anwenden muss und in [9] wird dieses Schema dann auf P2P Netzwerke angewendet.

Damit ein globaler Angreifer (passiv und aktiv) und selbst der RS nicht weiß, von wem die zu veröffentlichenden Daten tatsächlich kommen und welche Knoten der Gruppe nur Dummy-Nachrichten schicken, werden alle Nachrichten der Gruppe DC-net-artig mittels Onetime-Pads mit XOR-Funktion verschlüsselt, sodass die echten Daten nur extrahiert werden können, wenn alle Nachrichten empfangen und miteinander verknüpft wurden.

In dieser Masterarbeit wird der Aspekt der Verschlüsselung nicht beachtet, es wird lediglich die Erzeugung einer Gruppe und die anschließende Extraktion der echten Message aus den Cover Messages betrachtet. Die ADC-net-artige Verschlüsselung kann in einer weiteren Arbeit hinzugefügt werden und wurde beim Design des Prototypen explizit mit beachtet.

Auch ohne diese Verschlüsselung bleibt der Effekt erhalten, dass mehrere Knoten auf direkten Verbindungen Covertraffic zu einem RS erzeugen, welcher die zu publizierenden Datennachrichten dann als Stellvertreter im P2P Netzwerk veröffentlicht. Ohne Verschlüsselung kann ein passiver globaler Angreifer allerdings die Dummy-Nachrichten von den echten Datennachrichten unterscheiden. Die einfache Anwendung von Verschlüsselung, ohne eine Änderung des restlichen Algorithmus, schützt das System dann auch effektiv gegen diesen Angreifer.

5 Probleme und Lösungen

Versucht man die in Kapitel 4 geschilderten Konzepte ohne Veränderungen praktisch umzusetzen, so stößt man auf einige Probleme, die eine solche Umsetzung verhindern. Diese Probleme rühren einerseits von Unzulänglichkeiten im Entwurf her, sind andererseits aber auch dem Unterschied zwischen einer Simulation und einer echten Implementierung geschuldet. Die Unzulänglichkeiten im Entwurf mögen daher rühren, dass das Format „Paper“ nicht genügend Raum bietet, um alle Gedanken zum kompletten Entwurf im Detail darzulegen. Ich habe jedoch nur diese Paper [1, 10, 2] als Quelle vorliegen und kann das anhand der mir vorliegenden Daten deshalb nicht beurteilen.

Die Probleme und meine Vorschläge zur Lösung will ich im vorliegenden Kapitel darlegen. Die Details der Implementierung werde ich dann in Kapitel 6 schildern.

5.1 Hashketten-basiertes Routing von Daubert et al.

Das Hashketten-basierte Routing, so wie es in Abschnitt 4.3 beschrieben wurde, ist praktisch nicht direkt umsetzbar, da es mehrere ungelöste Probleme aufweist, die ich hier einzeln gemeinsam mit meinem jeweiligen Lösungsvorschlag erläutern will.

Das dabei neu entstandene System werde ich in Abschnitt 5.1.8 noch einmal detailiert beschreiben. Das dort geschilderte System ist, anders als die Konzepte aus Abschnitt 4.3, auch die Grundlage meiner Implementierung.

5.1.1 Problem 1: Mehrere Publisher

Daubert et al. beschreiben den Fall, dass mehr als nur ein Publisher für ein gegebenes Topic existiert, wie folgt: “When the system contains multiple publishers for a, every publisher $p \in P_a$ that receives an advertisement for a subscribes to it. This ensures the connectivity of overlay M_a.” [1, S. 47]

Entsteht ein zweiter Publisher p_2 für das gleiche Topic, für das ein anderer Publisher p_1 schon Advertisements in das Overlay geflutet hat, so sendet p_2 eine Subscription zum schon vorhandenen Publisher p_1, damit dadurch das Attribute Overlay verbunden bleibt. Es entsteht ein gerichteter Pfad von p_1 zu p_2 [10, S. 689].

Verlässt ein Knoten das P2P Overlay, so wird das Attribute Overlay von den verbleibenden Knoten des Attribute Overlays mittels *Unsubscribe* und *Unadvertise* Nachrichten repariert [10, S. 689, 1, S. 48].

Das erklärt allerdings nicht, wie ein nachfolgender Publisher p_2 seine Informationen den Subscribern zugänglich machen soll. Eine Subscription von p_1 bei p_2 erzeugt im Attribute Overlay die schon erwähnte gerichtete Kante von p_1 nach p_2, danach erhält p_2 die veröffentlichten Informationen von p_1, die er nicht benötigt. Seine eigenen Informationen empfangen die Subscriber deshalb trotzdem noch nicht.

Die Aussage, dass das Overlay durch die geschilderte Subscription von p_2 bei p_1 wieder vollständig verbunden ist, ist zwar im Ansatz korrekt, allerdings wird dabei unterschlagen, dass das Attribute Overlay

mit gerichteten Kanten operiert, die immer von einem Publisher zu einem Subscriber zeigen. Der Attribute Overlay hat die Funktion, Datennachrichten von den Publishern zu den Subscribern zu transportieren, eine Kante in die falsche Richtung ist daher für diese Funktion nutzlos.

Es wird in [1, 10] außerdem nicht erklärt, ob nachfolgende Publisher ebenfalls das P2P Overlay mit Advertisements fluten oder nicht.

Eine Rückfrage bei Daubert et al. ergab, dass die Subscription von p_2 bei p_1 nur für die von ihm durchgeführte Simulation verwendet wurde, da hierdurch die verfolgten Evaluationsziele einfacher erreicht werden konnten. Die Simulation hat außerdem noch eine Subscription von p_1 bei p_2 ausgelöst, sodass eine bidirektionale Kante entstand. p_2 hat dann seine Informationen entlang dieser Kante zu p_1 geschickt, der diese dann in Richtung der Subscriber veröffentlicht hat. Eine solche wechselseitige Subscription ist in der Praxis aber nicht nötig und erzeugt nicht nur eine höhere Netzwerklast (p_2 bekommt Daten von p_1, die er nicht benötigt), sondern auch unnötige Komplexität.

Lösung

Der nachfolgende Publisher p_2 hat den nächsten Schritt auf dem kürzesten Pfad zu p_1 bereits in seiner Routingtabelle der Advertisements stehen, er und die Zwischenknoten auf dem Pfad zu p_1 können daher diese Routingtabelle verwenden, um die Datennachricht von p_2 zu p_1 zu transferieren, aktive Pfade sind nicht nötig.

Diese Lösung hat den Vorteil, dass (anders als bei der Verbindung von Publishern und Subscribern), bei veränderten Netztopologien keine erneute Subscription und damit Aktivierung eines Pfades erfolgen muss, da die Publish Nachricht von p_2 automatisch den kürzesten verfügbaren Pfad zu p_1 nimmt, selbst wenn sich dieser Pfad durch neue Advertisements und/oder Knotenfehler verändert. Schleifen können dabei nicht entstehen, da die zugrunde liegenden Advertisement Routingtabellen schleifenfrei sind.

Begriffsdefinition 5.1:
Den (zeitlich) ersten Publisher p_1 für ein Attribut a (d.h. ein Topic bzw. ein Channel) im gesamten P2P Overlay nenne ich *Master-Publisher*, alle anderen Publisher $p_{2..n}$ für Attribut a dagegen *Slave-Publisher*.

Nur der Master-Publisher ist über aktive Pfade direkt mit den Subscribern verbunden. Er bekommt die Informationen von den Slave-Publisher, um sie stellvertretend für diese zu veröffentlichen.

5.1.2 Problem 2: Knotenfehler eines Master-Publishers

Ein P2P Netzwerk ist nicht statisch, sondern ist beständigem Wechsel unterzogen: neue Knoten betreten das Netzwerk, alte Knoten verlassen es. Knoten beginnen neu ein bestimmtes Topic zu publishen oder ein neues Topic zu subscriben.

Zu solchen Veränderungen gehört es auch, dass ein bisheriger Master-Publisher aufhört zu publishen. Das kann daran liegen, dass er das P2P Netzwerk komplett verlässt (möglicherweise ohne Vorwarnung, weil seine Netzwerkverbindung plötzlich abbricht), oder daran, dass er keine Informationen mehr veröffentlichen will. Geschieht dies, sind die Slave-Publisher von den Subscribern abgeschnitten, die sie nur über den Master-Publisher erreichen konnten. Das Attribute Overlay des Topics a ist somit komplett funktionsuntüchtig.

Bei Daubert et al. bemerken die Nachbarknoten des Master-Publishers zwar den Ausfall dieses Knotens und fluten Unadvertise Messages im P2P Overlay, es wird aber nicht erläutert, wie das defekte Attribute Overlay danach wieder funktionstüchtig gemacht werden kann. Daubert et al. schreiben lediglich, dass die Unadvertisements die Einträge des Master-Publishers aus allen Advertisement Routingtabellen entfernen: "Nodes leaving the attribute overlay send unadvertise and unsubscribe messages to their neighbors. Likewise, a node detecting a neighbor's failure acts like having received such a message from the neighbor. This ensures that obsolete routing table entries are purged and replaced with alternative entries." [1, S. 47]

Lösung

Haben die Slave-Publisher keinen Pfad zum Master-Publisher mehr in ihren Advertisement Routingtabellen stehen, so muss einer der Slaves selbst zum neuen Master werden.

Um zu verhindern, dass alle Slaves gleichzeitig zu Master-Publishern mutieren, muss jeder Slave eine zufällige Zeit warten, bevor er zum neuen Master wird und den Overlay mit Advertisements flutet. Hierbei lässt sich allerdings nicht vollständig verhindern, dass in manchen Fällen trotzdem mehrere Slave-Publisher zu neuen Mastern werden (siehe hierzu auch Abschnitt 5.1.3).

5.1.3 Problem 3: Mehrere Master-Publisher

Nicht nur wenn der originale Master-Publisher ausfällt und daraufhin mehrere Slaves zu neuen Mastern mutieren, kann es dazu kommen, dass mehrere Master für das gleiche Topic existieren. Auch beim initialen Verbinden des P2P Overlays lässt sich nicht vollständig vermeiden, dass mehrere Master-Publisher entstehen, da dieses verteilte System keine zentrale Registrierungsstelle für Publisher hat.

Prüfen beispielsweise alle Publisher direkt nach Eintritt in den P2P Overlay, ob sie für „ihr" Topic schon ein Advertisement empfangen haben, so muss das Fehlen eines solchen Advertisements nicht bedeuten, dass es keinen anderen Master-Publisher für dieses Topic gibt, sondern nur, dass ein solches Advertisement noch nicht bis zu diesem anderen Publisher vorgedrungen ist, also das Fluten noch nicht abgeschlossen ist.

Beim initialen Aufbau des P2P Overlays, bei dem sich die Knoten miteinander verbinden, würde das bedeuten, dass (fast) jeder Publisher direkt zu einem Master wird, da die Information über etwaige andere Master-Publisher noch nicht die Gelegenheit hatte, überall durch den Overlay zu fluten.

Lösung

Alle angehenden Publisher warten eine zufällige Zeit (Ich schlage etwa 4 bis 10 Sekunden vor), bevor sie prüfen, ob es mittlerweile einen (neuen) Master gibt. Ist das nicht der Fall, dann werden sie selbst zum Master. Dieses Vorgehen ist ähnlich zu CSMA (siehe [30]) und sorgt dafür, dass nicht zu viele Master entstehen.

Da man aber nie völlig verhindern kann, dass es mehrere Master gleichzeitig gibt, muss das System so ausgelegt werden, dass es auch mit mehreren gleichzeitig im P2P Overlay existierenden Mastern des gleichen Topics zurechtkommt. Dafür schlage ich vor, den üblichen Konzepten zu folgen, sodass

sich die Subscriber immer mit allen verfügbaren/bekannten Publishern verbinden (d.h. einen aktiven Pfad von diesen Publishern zu den Subscribern erzeugen). Die verschiedenen Master können dabei von den Subscribern über ihre unterschiedlichen Transaktionspseudonyme (d.h. die Zugehörigkeit zu verschiedenen Hashketten) eindeutig unterschieden werden.

Alle Master-Publisher sind in diesem System völlig gleichberechtigt. Da jeder Subscriber mit allen Mastern verbunden ist, können die Slaves sich aussuchen, zu welchem der ihnen bekannten Mastern sie ihre Informationen publishen. Hierbei ist auch die Möglichkeit gegeben, für jedes Publish einen zufälligen Master zu wählen. Fällt einer der Master oder ein Knoten auf dem kürzesten Weg zu diesem Master aus, so gehen nicht alle Nachrichten verloren, bis das durch den Knotenfehler ausgelöste Unadvertisement den Overlay repariert hat. Der Begriff „repariert“ bedeutet in diesem Zusammenhang, dass die Advertisement Routingtabellen aller Knoten wieder konsistent sind und den tatsächlichen Status des Netzwerks widerspiegeln. Das kann jedoch dafür sorgen, dass die Nachrichten wesentlich häufiger in einer falschen Reihenfolge ankommen, als das mit einem fest ausgewählten Master der Fall wäre.

Welche Übertragungsart der Slaves man wählt, hängt stark vom Anwendungsfall und den zu übermittelnden Daten ab. Möchte man beispielsweise ein Telefongespräch übertragen oder auch eine ganze Telefonkonferenz, ist die Wahl eines festen Masters besser, da die Datennachrichten zu den verschiedenen Mastern sehr unterschiedliche Laufzeiten haben können. Diese unterschiedlichen Laufzeiten sorgen dafür, dass die Reihenfolge der ankommenden Datennachrichten sich stark von der ausgehenden Reihenfolge unterscheidet. Dadurch wird die Qualität der Echtzeitkommunikation herabsetzt. Sollen dagegen nicht echtzeitkritische Sensorwerte, beispielsweise die Temperaturen von Wetterstationen, übermittelt werden, für die die Reihenfolge der ankommenden Messwerte nicht relevant ist, so können die Slaves für jede Datennachricht einen zufälligen Master wählen. Das verringert den Verlust von Messwerten, falls ein Master ausfällt.

Allgemein lässt sich als grobe Richtlinie festlegen, dass echtzeitkritische Anwendungen von der Verwendung eines festen Masters profitieren können, während andere eher von zufällig gewählten Mastern und dem damit einhergehenden potentiell geringeren Nachrichtenverlust profitieren, solange die Reihenfolge der ankommenden Datennachrichten für den Anwendungsfall nicht relevant ist.

5.1.4 Problem 4: Nachträglich hinzugefügte Knoten

Daubert et al. erklären nur den initialen Aufbau eines Attribute Overlays in einem fertig verbundenen P2P Netzwerk. Die Einbindung neuer Knoten, die nachträglich dem P2P Overlay beitreten, wird nicht erklärt.

Solche neuen Knoten können sowohl die Rolle eines Forwarders, als auch die eines Subscribers oder Publishers einnehmen. Diese neuen Knoten sind von allen Attribute Overlays abgeschnitten, für die bereits Publisher bestehen. Sind diese neuen Knoten Publisher für ein Topic, für das bereits andere Publisher existieren, so werden sie trotzdem zu Mastern.

Lösung

Neue Knoten müssen von ihren Nachbarknoten die Einträge der Advertisement Routingtabelle geschickt bekommen und diese in ihre eigenen Routingtabellen übernehmen.

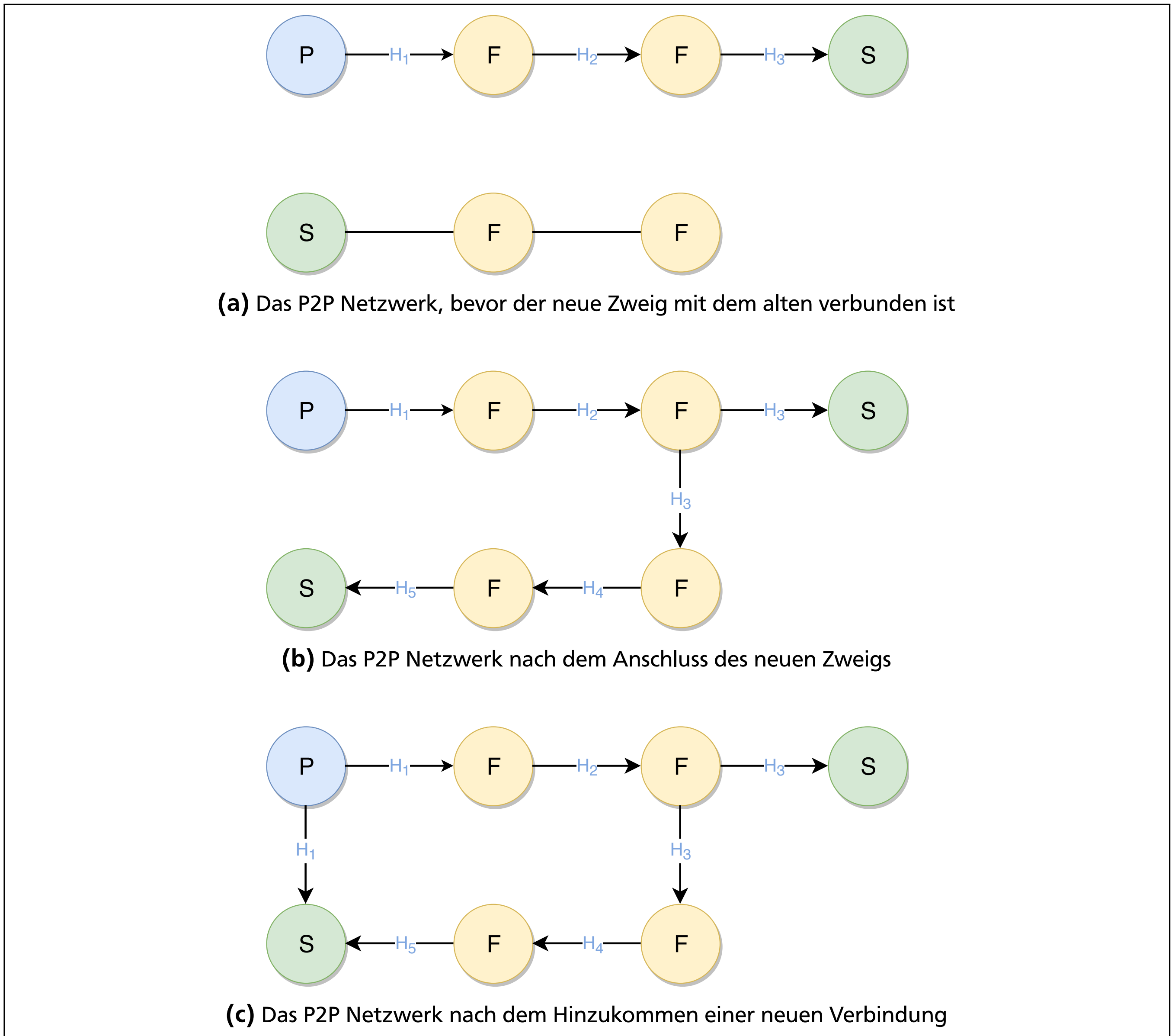


Abbildung 5.1.: Reflooding der Advertisements bei neu zum P2P Netzwerk hinzukommenden Knoten und Verbindungen

Da es durchaus sein kann, dass die neuen Knoten bereits ebenfalls neue Nachbarn haben, die sonst keine weitere Berührungsstelle mit dem restlichen P2P Netzwerk haben, müssen die solcherart ausgetauschten Einträge der Routingtabelle auch an diese Nachbarknoten weiter geflutet werden (siehe Abbildung 5.1). Allerdings nur so lange, bis die empfangenden Knoten bereits einen Eintrag der gleichen Hashkette in ihrer Routingtabelle haben (selbst wenn dieser länger sein sollte, siehe Abbildung 5.1c).

Hierdurch wird verhindert, dass sich das bereits aufgebaute Attribute Overlay dauernd verändert bzw. neu aufbaut, nur weil neue Knoten das P2P Overlay betreten haben und damit neue (kürzere) Pfade anbieten, da erwartet wird, dass ein solcher ständiger Umbau und die damit einhergehenden Management Messages den Gewinn zunichte machen würden, den die neuen kürzeren Pfade bieten würden.

5.1.5 Problem 5: Konkurrierende Advertisements

Die Advertisements, die einen Subscriber oder auch Forwarder erreichen, kommen nicht notwendigerweise immer in der Reihenfolge „kürzer“ vor „länger“ an. Die Rechengeschwindigkeit eines Knotens auf der kürzesten Strecke zum Subscriber oder eine hohe Netzwerklast auf dieser Strecke bzw. Teilen der Strecke, können dafür sorgen, dass beim Subscriber zuerst ein längeres Advertisement ankommt oder sogar gleich mehrere längere, bevor dann das eigentlich gewünschte kürzeste Advertisement den Subscriber erreicht.

Reagiert der Subscriber sofort mit einer Subscription, sobald er das erste Advertisement erhält, sorgt das unter Umständen dafür, dass nicht der kürzeste Pfad vom Publisher zum Subscriber aktiviert wird. Das hängt mit der Topologie des P2P Netzwerks und des verwendeten Underlays (Internet etc.) zusammen. Je nachdem, wie groß der Jitter oder Paketverzögerungen/Paketverluste auf bestimmten Verbindungen sind, kann dieses Problem größer oder kleiner sein. Zu erwarten ist allerdings, dass sich die Länge des dann verwendeten Pfades nicht sonderlich stark vom kürzesten Pfad unterscheidet.

Lösung

Durch eine einfache Lösung lässt sich dieser Effekt jedoch abmildern und es ist zu erwarten, dass er fast vollständig kompensiert werden kann.

Die Lösung besteht darin, eine gewisse Zeitspanne zu warten, ob nicht noch weitere kürzere Advertisements ankommen, bevor der Subscriber dann seine Subscription abschickt und damit einen aktiven Pfad vom Publisher zum Subscriber erzeugt (für die Erklärung aktiver Pfade, siehe Abschnitt 4.5).

Diese Zeitspanne ist in meiner Implementierung auf zwei Sekunden voreingestellt. Eine höhere Wartezeit sorgt dafür, dass auch mehr Zeit vergeht, bis ein Subscriber dann tatsächlich Daten von den Publishern empfängt, eine niedrigere Wartezeit erhöht die Wahrscheinlichkeit, nicht die optimalen Pfade zum Publisher zu aktivieren. Hier muss ein Tradeoff gefunden werden, der auch von der Anzahl der Knoten im P2P Netzwerk abhängt. Eine Evaluation von verschiedenen Zeitspannen kann in Abschnitt 7.2.2 gefunden werden.

5.1.6 Problem 6: Knotenfehler und alternative Pfade

Daubert et al. schreiben, dass bei einem Knotenfehler die benachbarten Knoten stellvertretend für den ausgefallenen Knoten *Unadvertise* und *Unsubscribe* Nachrichten versenden sollen [10]. Ein Hinweis, wie die Konnektivität des Attribute Overlays wiederhergestellt werden kann, wenn der ausgefallene Knoten ein Forwarder war, geben die Autoren hier: “Nodes leaving the attribute overlay send unadvertise and unsubscribe messages to their neighbors. Likewise, a node detecting a neighbor’s failure acts like having received such a message from the neighbor. This ensures that obsolete routing table entries are purged and replaced with alternative entries.” [1, S. 47] Woher diese alternativen Einträge der Routingtabelle kommen sollen, wird aber nicht erklärt.

Da die Nachbarknoten eines Publishers oder Forwarders, der weiter entfernt von diesem Publisher ist, nicht wissen können, wie nah sie tatsächlich am Publisher sind, müssen alle diese Knoten Unadvertisements fluten, wenn der Nachbarknoten, auf den ihr Eintrag in der Routingtabelle verweist, nicht mehr erreichbar ist. Ist jedoch nur ein Forwarder auf dem Weg zum Publisher ausgefallen und nicht der Publisher selbst, so muss sich das System selbst heilen und einen alternativen Pfad zum Publisher finden.

Lösung

Empfängt ein Knoten einen gleich langen oder längeren Hash als den bereits bekannten einer Kette, so flutet er das Advertisement nicht mehr an seine Nachbarknoten weiter, trägt aber auch diesen längeren Hash in seine Routingtabelle ein. Das sorgt dafür, dass bei späteren Knotenfehlern Alternativen gefunden werden können, verhindert aber trotzdem, dass Schleifen gebildet oder Duplikate über das gesamte P2P Overlay geflutet werden. Wird dagegen ein kürzerer Hash empfangen, als der bereits bekannte, oder ist die Hashkette noch nicht bekannt, so wird der Hash nicht nur in der Routingtabelle gespeichert, sondern das Advertisement auch weiter geflutet.

Das Speichern mehrerer Einträge der selben Hashkette in der Routingtabelle muss dabei derart geschehen, dass auch später noch der kürzeste dieser Einträge gefunden werden kann. Dieser Eintrag ist dann der *aktive*.

Beim Empfang eines Unadvertisements mit dem Tupel (H, v) müssen dann alle Einträge aus der Routingtabelle entfernt werden, die diesem Tupel entsprechen. Hierbei steht v für den Nachbarknoten, von dem das Unadvertisement oder Advertisement kam und H für die Hashkette aus dem Unadvertisement bzw. Advertisement (Achtung: hiermit ist *nur* die Hashkette gemeint, nicht der konkrete Hash).

Muss dabei kein einziger Eintrag aus der Routingtabelle entfernt werden, wird das Fluten des Unadvertisements hier abgebrochen, da die Hashkette nicht bekannt ist und daher vorher auch nicht durch ein von diesem Knoten weitergeleitetes Advertisement bei den Nachbarknoten bekannt gemacht worden sein kann.

Ist nach dem Entfernen der Einträge aus der Routingtabelle noch mindestens ein Eintrag der betrachteten Hashkette vorhanden, dann wird der kürzeste dieser Einträge zu allen Nachbarknoten geflutet, da diese durch das vorhergehende Fluten des Unadvertisements nun keinen Eintrag dieser Hashkette mehr in ihrer Routingtabelle stehen haben, außer sie kennen noch einen alternativen Pfad. Bei diesem Fluten eines neuen Advertisements wird der Knoten ausgenommen, auf den dieser Eintrag der Routingtabelle zeigt. Dieses Fluten muss außerdem, wie in Abschnitt 5.1.4 beschrieben, automatisch an denjenigen Knoten stoppen, die bereits einen anderen Eintrag der Hashkette in ihrer Routingtabelle haben, egal ob dieser Eintrag länger oder kürzer ist.

Mit diesem Verfahren wird sichergestellt, dass alle Knoten, die nach dem Unadvertisement keinen Pfad mehr zum Publisher kennen, eine neue Alternative mitgeteilt bekommen, während andere Teile des Netzwerks, die nicht von diesem Ausfall betroffen sind, davon keine Notiz nehmen.

Fällt tatsächlich der Publisher selbst aus, werden durch das Fluten des Unadvertisements alle alternativen Pfade abgebaut, sodass der Hash dieses Publishers danach bei keinem einzigen Knoten des P2P Netzwerks mehr in der Routingtabelle steht. In diesem Fall muss dann ggf. ein neuer Master gefunden werden (siehe Abschnitt 5.1.2 und Abschnitt 5.1.3).

5.1.7 Problem 7: Count to Infinity

Durch das in Abschnitt 5.1.6 vorgeschlagene Fluten von alternativen Pfaden nach aufgetretenen Knotenfehlern kann es passieren, dass Schleifen im P2P Netzwerk eine *Count to Infinity* Problematik entwickeln, ähnlich wie dies bei Distanzvektoralgorithmen der Fall ist [31]. Das liegt daran, dass die Knoten einer solchen Schleife noch Alternativen über ihren anderen Nachbarknoten der Schleife kennen und diesen

direkt beim Empfang eines Unadvertisements als neues Advertisement an den Nachbarn schicken, von dem das Unadvertisement kam.

Abbildung 5.2 verdeutlicht dieses Szenario. Bei einem Ausfall des Publishers würde beispielsweise der Forwarder F_1 ein Unadvertisement an den Forwarder F_2 schicken, der dieses an seine Nachbarn F_3 und F_4 weiterschickt. Diese würden zum Zeitpunkt des Empfangs aber noch einen weiteren Pfad zum gleichen Publisher kennen, der über ihre Nachbarknoten F_3 bzw. F_4 läuft und diesen Pfad als Alternative mit dem Hash H_5 an F_2 zurückfluten.

Bestehen die Schleifen aus nur einigen wenigen Knoten (beispielsweise drei), so stabilisiert sich die Schleife nach einigen Schritten wieder und die Advertisements der Alternativen laufen nicht endlos im Kreis. Allerdings steht danach auch in diesem Fall die Hashkette des Publishers P in der Routingtabelle der Knoten, obwohl dieser das P2P Netzwerk längst verlassen hat.

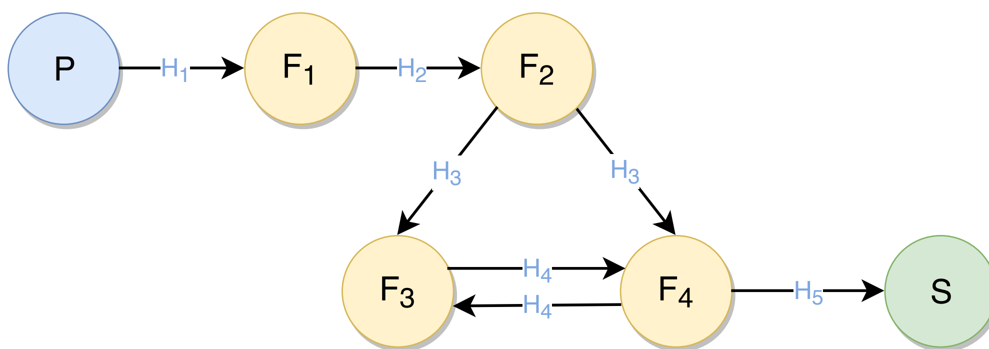


Abbildung 5.2.: Count to Infinity bei Hashketten-basiertem Routing

Lösung

Dieses Problem lässt sich lösen, indem die Knoten nach dem Empfang eines Unadvertisements, ähnlich wie die Subscriber in Abschnitt 5.1.5, eine Zeit lang warten, bevor sie neue Advertisements an ihre Nachbarn fluten, die die ihnen dann noch bekannten Pfade enthalten.

Hierdurch kann sich das Unadvertisement in der kompletten Schleife ausbreiten, bevor die Knoten dieser Schleife prüfen, ob sie noch Alternativen kennen und diese an ihre Nachbarknoten weitergeben.

Diese Wartezeit muss abhängig vom Durchmesser des P2P Netzwerks groß genug gewählt werden, sodass die Unadvertisements überall in der Schleife ankommen, bevor ein Re-Advertisement stattfindet. Da die Mechanismen beim Fluten von Unadvertisements die gleichen sind wie beim Fluten von Advertisements, wird erwartet, dass die besten Ergebnisse mit einer Wartezeit erzielt werden, die mindesten so groß ist, wie die Wartezeit, die in Abschnitt 5.1.5 vorgeschlagen wurde.

Selbstverständlich lassen sich auch komplexere Lösungen für diese Problematik entwickeln. In Anbetracht der an anderen Stellen des Systems bereits ebenfalls stattfindenden Wartezeiten halte ich die hier vorgeschlagene Lösung aber für ausreichend.

5.1.8 Zusammenfassung der Änderungen am System

Fasst man die obigen Lösungen der Probleme zusammen, entsteht das folgende Bild des (neuen) Hashketten-basierten Routings. In dieser Zusammenfassung wird keine Begründung einzelner Vorgehensweisen gegeben, sondern nur eine reine Prozessbeschreibung geliefert. Für eine Begründung verweise ich dagegen auf die Abschnitte 4.3 und 5.1.

Master-Publisher:

Jeder Publisher wartet eine gewisse Zeit, bevor er prüft, ob für sein Topic bereits ein anderer Master-Publisher bekannt ist. Existiert ein solcher Master, dann wird der Publisher zum Slave, existiert kein anderer Master, so wird der Publisher selbst zum Master und flutet den P2P Overlay mit Advertisements. Die Prüfung, ob bereits ein anderer Master für ein Topic existiert, erfolgt anhand der eigenen Advertisement Routingtabelle. Es kann also sein, dass ein weit entfernter Publisher bereits ebenfalls begonnen hat das P2P Overlay mit Advertisements zu fluten, diese aber nur noch nicht bei dem prüfenden Publisher angekommen sind. In diesem Fall wird es mehr als einen Master-Publisher geben.

Fluten von Advertisements:

Die Advertisements der Master-Publisher enthalten einen Hash, der bei jeder Weiterleitung des Advertisements erneut gehasht wird, sodass sich eine Hashkette bildet. Ebenfalls enthalten ist das Topic und die Information, ob es sich um ein erneutes Fluten handelt, welches von einem unbeteiligten Knoten ausgeht, oder ob das Fluten vom Master selbst initiiert wurde. Handelt es sich um das Fluten eines Masters, so wird auch dann weiter geflutet, wenn die Hashkette bereits bekannt ist, aber das Advertisement kürzer ist als der bereits bestehende Eintrag in der Routingtabelle. Kommt das Fluten von einem unbeteiligten Knoten, wird dieses Fluten dagegen sofort beendet, sobald ein Knoten die Hashkette aus dem Advertisement bereits kennt.

Möchte ein Master nichts mehr publishen, so flutet er ebenfalls Unadvertise Messages, die dafür sorgen, dass die Subscriber die aktiven Pfade zu ihm abbauen.

Aufbau der Routingtabellen:

Empfängt ein Knoten ein Advertisement, so übernimmt er es in seine Routingtabelle indem er den Empfangsknoten zusammen mit dem Topic und dem Hash aus dem Advertisement speichert. Hat er bereits einen Eintrag für diese Hashkette in seiner Routingtabelle, dann vergleicht er die Länge der beiden Hashes. Nur wenn das Advertisement kürzer ist, wird es weitergeschickt. Auf diese Weise haben zwei benachbarte Knoten, die beide auf unterschiedlichen kürzesten Pfaden zum Publisher liegen, jeweils zwei Einträge in ihren Routingtabellen: den kürzesten Pfad und einen um maximal Eins längeren alternativen Pfad über ihren Nachbarn (siehe auch Abbildung 5.3).

Slave-Publisher veröffentlichen ihre Daten entlang des kürzesten Pfades der Advertisement Routingtabelle zu einem ihnen bekannten frei wählbaren Master, der die Daten dann zu den Subscribern weiterleitet.

Subscriber:

Empfängt ein Subscriber ein Advertisement für das ihn interessierende Topic oder wird ein Knoten zu einem Subscriber für ein bereits in der Advertisement Routingtabelle stehendes Topic, so wartet er einige Sekunden, bevor er Subscription Messages entlang der aktuell bekannten kürzesten Pfade zu allen ihm bekannten Master-Publishern schickt.

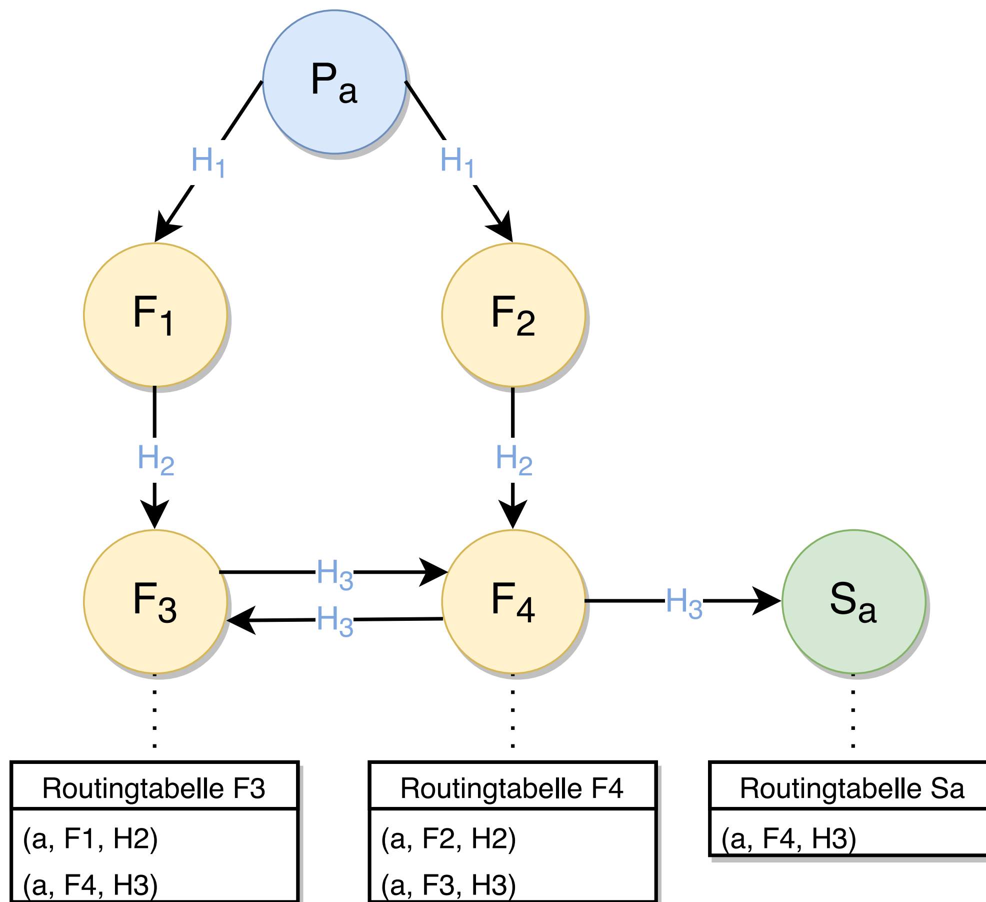


Abbildung 5.3.: Beispiel der Routingtabellen beim Hashketten-basierten Routing

Die Subscription Message sorgt dafür, dass alle Knoten auf dem Pfad zu dem in der Message spezifizierten Master den Pfad der Subscription Message als aktiv markieren, d.h. dem Attribute Overlay hinzufügen, sodass der Master hierüber Daten an diesen Subscriber veröffentlichen kann.

Knotenausfälle:

Fällt ein Master-Publisher aus, wird dieser Ausfall durch Unadvertise Messages im P2P Overlay an alle Knoten geflutet. Kommt die Information über den ausgefallenen Master bei einem Slave an, wartet dieser eine gewisse Zeit, bevor er erneut prüft, ob bereits ein Advertisement eines neuen Masters angekommen ist. Ist das nicht der Fall, wird er selbst zum neuen Master.

Fällt ein Forwarder aus, werden von den Nachbarknoten ebenfalls Unadvertise Messages versendet, wenn sie einen Pfad über diesen Knoten kannten. Knoten, die noch einen alternativen Pfad kennen, fluten das Unadvertisement nur zu diesen alternativen Nachbarknoten, damit diese den Knoten selbst als Alternative aus ihren Routingtabellen entfernen können, da an der Schnittstelle zweier Pfade die beiden involvierten Nachbarknoten den jeweils anderen Pfad kennen.

Nach einer Wartezeit senden diejenigen Knoten, die noch einen alternativen Pfad kennen und bei denen ein Unadvertisement vorbei kam, neue Advertisements an ihre Nachbarknoten. Diese Advertisements werden aber von Knoten, die bereits Einträge dieser Hashkette in ihren Routingtabellen stehen haben *nicht* weitergeleitet, selbst wenn das Advertisement kürzer sein sollte, sodass nur die Knoten, die keinen einzelnen Pfad mehr zum Publisher kennen mit neuen Pfadinformationen versorgt werden.

Nachträglich zum P2P Overlay hinzukommende Knoten:
Verbindet sich ein Knoten neu mit dem P2P Overlay, so bekommt er von seinen Nachbarn die Routingtabellen geschickt, sodass er ohne erneutes Fluten von Advertisements vollständig in das Netzwerk integriert ist. Diese neuen Knoten schicken dann an alle ihre Nachbarn Advertisements, um diese ebenfalls in das Netzwerk einzugliedern. Auch hier wird die Weiterleitung eines Advertisements gestoppt, sobald ein Knoten bereits einen Eintrag in seiner Routingtabelle vorfindet, selbst wenn der neue Eintrag kürzer sein sollte.

5.2 ACO Routing

Auch das ACO-basierte Routing aus [2] lässt sich ohne kleinere Anpassungen nicht direkt implementieren. Die nötigen Anpassungen sind hier allerdings größtenteils dem Unterschied zwischen Simulation und echter Implementierung geschuldet.

Im Folgenden will ich die nötigen Anpassungen daher kurz vorstellen und erklären. Eine Zusammenfassung der Änderungen ist bei ACO allerdings nicht nötig, da die Anpassungen überschaubar bleiben und das im Paper geschilderte Verhalten nicht tiefgreifend ändern.

5.2.1 Umstellung von rundenbasiert auf zeitbasiert

Der Algorithmus 1 aus [2] ist rundenbasiert. Das ist gerade für Simulationen ein großer Vorteil, da hierdurch eine größere Strukturiertheit entsteht und Probleme aus Nebenläufigkeiten umgangen werden können, ohne dass der Grundgedanke des Algorithmus dadurch beeinflusst wird.

Möchte man aber den Aufbau eines Attribute Overlays mittels ACO auf ein echtes verteiltes System bestehend aus mehreren Knoten übetragen, muss der rundenbasierte Algorithmus in einen zeitbasierten transformiert werden, da sonst eine zentrale Instanz notwendig wäre, die den Beginn und das Ende einer Runde vorgibt.

Die Transformation auf einen zeitbasierten Algorithmus lässt sich über die Verwendung eines Rundentimers erreichen. Jeder Subscriber, der Ameisen ausschickt, startet demnach in einem festgelegten Intervall eine neue Ameisenrunde, in der er eine festgelegte Menge von Ameisen aussendet. Ich schlage vor, für dieses Intervall etwa fünf bis zehn Sekunden zu verwenden und in jeder Runde etwa fünf Ameisen auszusenden. Diese Zahlen sollten allerdings nur als Richtwerte angesehen werden.

Das Verdampfen der Pheromone, welches in [2] noch fest am Anfang jeder Runde stattfindet, muss ebenfalls zeitgesteuert sein. Ich schlage dafür in Einklang mit dem Rundenintervall ebenfalls fünf bis zehn Sekunden vor.

5.2.2 Keine reale Entsprechung der Entität „Kante“

In einer realen Implementierung gibt es die Entität „Kante“ nicht, sondern nur Knoten. Sollen Pheromone auf eine Kante gelegt werden, müssen sie bei einem der Knoten oder sogar beiden Knoten gespeichert werden, die durch die Kante verbunden werden.

Die Pheromone müssen gerichtet sein (wie die aktiven Kanten auch), da sie die Ameisen immer zu einem Publisher führen sollen und nicht von diesem weg. Die Richtung der Pheromone ist dabei umgekehrt zur Richtung der aktiven Kanten.

Um die Pheromone solcherart gerichtet zu speichern, muss die Ameise diese auf dem Rückweg *nach* dem Traversieren einer Kante auf den damit erreichten Knoten legen. Der letzte Knoten, bei dem die Ameise ihre Pheromone daher auf eine Kante legt, ist der Subscriber der Ameise selbst.

5.2.3 Zahl der Ameisenrunden

Die durch die ACO optimierten aktiven Pfade konvergieren hin zu einer optimalen Lösung (siehe [20, 21]). Nach der Konvergenz ist es daher nicht mehr nötig, weiter Ameisen loszuschicken.

Der Status der Konvergenz kann nicht einfach gemessen werden, weil es dafür eine globale Referenz (beispielsweise einen Steinerbaum) geben müsste. Dadurch könnte allerdings die Anonymität des Systems nicht mehr gewahrt werden (siehe dazu Kapitel 2). Aus diesem Grund kann das Starten neuer Ameisenrunden nur nach einer bestimmten Zeit bzw. einer bestimmten Anzahl von Runden beendet werden. Ich schlage dafür in Einklang mit den Autoren des Papers 150 Runden vor. Da die Anzahl der Runden den maximal möglichen Abstand zwischen Publishern und Subscribern festlegt, sollte bei Netzwerken mit einem größeren Durchmesser eventuell eine höhere Zahl an Runden gewählt werden.

5.2.4 Erzeugung von aktiven Kanten

Damit die in Abschnitt 4.5 beschriebenen aktiven Pfade erzeugt werden können, müssen bestimmte Ameisen diese Pfade aktivieren. Ich schlage dafür in Einklang mit den Autoren des Papers vor, alle 50 Runden zwei Aktivierungsameisen loszuschicken (bei fünf Ameisen pro Runde wären das zwei Aktivierungsameisen und drei normale Ameisen). Die Begrenzung der Aktivierungsameisen auf zwei sorgt dafür, dass die aktiven Pfade in der Aktivierungsrunde nicht mehrfach auf- und abgebaut werden während gleichzeitig der durch Knotenfehler induzierte Verlust von zumindest einer der beiden Ameisen tolleriert werden kann. Ein mehrfaches Auf- und Abbauen der aktiven Pfade erzeugt Last im *Covert Channel*, dessen Datenübertragungsrate beschränkt ist. Diese Last sollte daher möglichst gering gehalten werden, ohne gleichzeitig die Funktion und Konvergenz von ACO dabei einzuschränken.

Die in Abschnitt 4.5 beschriebenen aktiven Pfade sind aus den dort geschilderten Gründen versioniert. Damit die Pfade korrekt versioniert werden können, müssen die Aktivierungsameisen streng monoton steigend durchnummeriert werden und diese Nummerierung dann als *Version* für den aktiven Pfad verwendet werden, den die Aktivierungsameise erzeugt.

5.2.5 Versteckte Publisher

In Algorithmus 1 von [2] und im Begleittext wird beschrieben, dass Ameisen zu ihrem Subscriber zurückkehren, sobald sie auf ihrem Randomwalk auf einen Publisher des gesuchten Topics treffen. Befindet sich aber ein weiterer Publisher *hinter* diesem ersten, so wird der weitere Publisher nicht von der Ameise gefunden und daher auch kein aktiver Pfad zu diesem aufgebaut (siehe Abbildung 5.4).

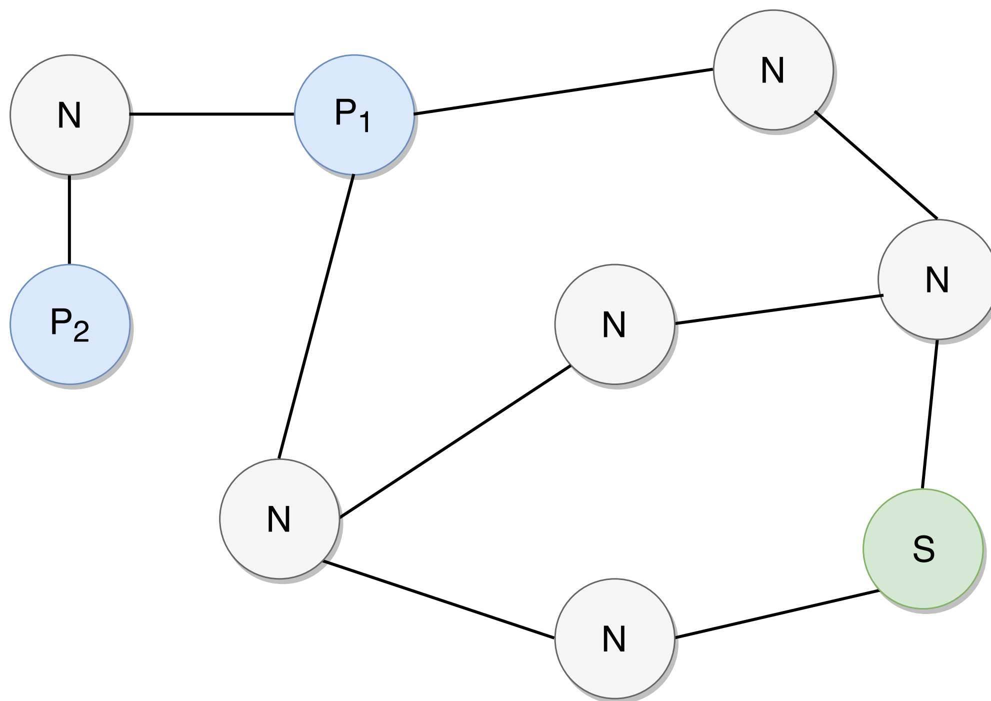


Abbildung 5.4.: Ein hinter Publisher P_1 versteckter Publisher P_2

Zur Lösung dieses Problems genügt es, die Ameise beim Erreichen eines Publishers zu klonen. Eine der Ameisen läuft dann zurück zum Subscriber, während die andere für die ihr verbliebene TTL weiter nach Publishern sucht.

5.2.6 Fluten einer Publish Message

Damit Subscriber, die bereits dem P2P Netzwerk beigetreten sind, neue Ameisenrunden starten können, wenn ein neuer Publisher im Netzwerk entstanden ist, müssen diese über diesen Sachverhalt informiert werden.

Im Paper ist dieser Fall nicht behandelt. Ich schlage deshalb vor, dass neue Publisher eine *Publish Message* im P2P Overlay fluten, die das für dieses Topic verwendete Pseudonym (siehe dazu auch Abschnitt 4.5) und die Bezeichnung des Topics enthält. Anhand der Topicbezeichnung können dann potentielle Subscriber entscheiden, ob sie Ameisenrunden starten, während das Pseudonym des Publishers zur Deduplizierung und Schleifenerkennung beim Fluten verwendet wird.

5.2.7 Overlay Maintenance

Aktive Pfade lassen sich periodisch erneuern, um damit auf veränderte bzw. verbesserte Netzwerktopologien und die damit eventuell einhergehenden kürzeren Pfade zu reagieren. Für eine solche Erneuerung starten die Subscriber wieder neue Ameisenrunden. Da zu erwarten ist, dass die Ameisenrunden eine recht hohe Last im *Covert Channel* erzeugen, sollte eine solche Maintenance nicht zu häufig gemacht werden.

Geht man von 150 Ameisenrunden und einem Rundenintervall von fünf Sekunden aus, so schlage ich eine Maintenance alle $(150 * 5) * 8 = 6000$ Sekunden (100 Minuten) vor. Ist die Fluktuation im P2P

Netzwerk recht gering, so wird erwartet, dass die Maintenance keine nennenswerten Verbesserungen bringt und gänzlich vermieden werden sollte, da diese in diesem Fall nur eine zusätzliche Belastung des *Covert Channels* darstellt.

6 Implementierung

In diesem Kapitel will ich mich der Implementierung der in Kapitel 4 und 5 vorgestellten Konzepte widmen. Dazu werde ich in Abschnitt 6.1 einen Überblick über das System geben, in Abschnitt 6.2 die Struktur des Frameworks erläutern, in Abschnitt 6.3 die webbasierte Oberfläche des Systems vorstellen und in Abschnitt 6.4 schlussendlich die Implementierung der Konzepte aus Kapitel 4 und 5 angesprochen.

6.1 Überblick

Das hier vorgestellte System versteht sich nicht als eine voneinander unabhängige Implementierung der einzelnen, in dieser Arbeit vorgestellten, Routingalgorithmen und Anonymisierungstechniken, sondern als ein Framework, welches es dem Nutzer ermöglicht neue Routingalgorithmen oder Anonymisierungstechniken schnell und einfach prototypisch zu entwickeln, ohne sich dabei um die Details der Netzwerkprogrammierung oder Multithreading/Locking Gedanken machen zu müssen.

Da die Sprache Python (neben einigen anderen Sprachen auch) ein einfaches und schnelles Prototyping ermöglicht, auf allen gängigen Plattformen (Windows, Linux und MacOS) verfügbar ist und der Autor bereits Erfahrungen mit dieser Sprache sammeln konnte, fiel die Wahl der verwendeten Programmiersprache auf Python in der Version 3 (3.5 oder höher).

Im Vergleich zu Java benötigt die Runtime von Python außerdem weniger Resourcen (speziell RAM). Da das System zu Vorführungszwecken auf mehreren Raspberry Pi Minicomputern laufen soll, die weniger RAM haben, ist Python auch in dieser Hinsicht eine gute Wahl.

Neben den Packages und Modulen der Python Standardbibliothek werden außerdem die folgenden Packages verwendet: *numpy* (Debian-Package: python3-numpy), *cryptography* (Debian-Package: python3-cryptography), *cherrypy3* (Debian-Package: python3-cherrypy3), *sortedcollections* (Debian-Package: python3-sortedcollections, nur in Debian Buster und neuer verfügbar), *sortedcontainers* (Debian-Package: python3-sortedcontainers) und *networkx* (Debian-Package: python3-networkx). Das Package *networkx* wird dabei nur für die Generierung von Graphen bei der Evaluation, nicht aber für den Betrieb des P2P Netzwerks benötigt.

6.2 Das Framework

Das Framework ist Multithreaded und besteht aus mehreren Komponenten, die untereinander an einigen Stellen mittels First-in-first-out (Fifo)-Queues kommunizieren und so voneinander entkoppelt sind. In einer solchen Queue kann jeder Thread einer Anwendung Daten ablegen, die dann in der gleichen Reihenfolge von anderen Threads aus der Queue heraus geholt werden können. Die Queue der Python Standardbibliothek ist dabei threadsafe implementiert, sodass diese Implementierung ohne weitere Vorkehrungen im Multithreading-Umfeld verwendet werden kann.

Die Komponenten des Packages **networking** legen bei auftretenden Ereignissen diese, in Form einer Dictionary eines vordefinierten Formats, in einer Router-Queue ab, sodass diese Ereignisse vom gestarteten Router (d.h. der Implementierung eines Routingalgorithmus) verarbeitet werden können.

Viele Stellen des Systems legen darüber hinaus auch Ereignisse in einer weiteren sogenannten Event-Queue ab. Dazu gehört neben den Logmeldungen des Systems auch der Bearbeitungszustand von Befehlen aus der Benutzeroberfläche, der Zustand der Netzwerkverbindungen des Systems und der Zustand der LEDs. Diese Events können von Benutzeroberflächen zur Anzeige genutzt werden. Ein Beispiel einer solchen Benutzeroberfläche stelle ich in Abschnitt 6.3 vor.

Lässt man das Framework auf einem Raspberry Pi Minicomputer laufen, so kann man dort LEDs anschließen, die sich so konfigurieren lassen, dass sie den Zustand des Systems bzw. des Routers anzeigen. Für die Ermittlung und Auswertung dieses Zustands wurde im Framework ein Filtersystem implementiert, welches es erlaubt, im laufenden Betrieb Filterdefinitionen in die Knoten des P2P Netzwerks zu laden. Neben der LED-Steuerung können diese Filter auch Logausgaben tätigen.

Die Router wiederum können vollkommen Event-basiert arbeiten und sind dadurch komplett lockfrei. Ankommende Nachrichten, neue oder abgebrochene Verbindungen und vom Router frei konfigurierbare Timer werden als Event in die bereits erwähnte Router-Queue gelegt, die von einem einzelnen Thread in der Router-Basisklasse verarbeitet wird. Die Klassenhierarchie versteckt diese Details vor der konkreten Implementierung eines Routers.

Für jeden Befehl aus der Router-Queue wird von der Basisklasse die dazugehörige Methode aufgerufen, welche den Befehl und die darin gespeicherten Daten übergeben bekommt. Den Routern stehen darüber hinaus einige Mixins zur Verfügung, mit denen sie oft benötigte Funktionen wie beispielsweise das Management von aktiven Pfaden nachrüsten können.

Durch all diese Funktionen des Frameworks, lässt sich die Implementierung eines einfachen Randomwalk-basierten Routers in weniger als 50 Zeilen Code realisieren (siehe Listing A.2).

Neben dem Filtersystem und der Event-Queue stellt das Framework auch eine vollständig Representational State Transfer (REST)-basierte Schnittstelle über Hypertext Transfer Protocol (http) bereit, über die das komplette System gesteuert werden kann. Die Events aus der Event-Queue werden von dieser Schnittstelle als Server Sent Events (SSE)-Events über http gestreamt, sodass diese beispielsweise einfach als Javascript-Events von einer Browser-basierten Oberfläche konsumiert werden können. Auch andere nicht Javascript-/Browser-basierte Oberflächen sind denkbar, da das Format eines SSE-Streams sehr einfach gehalten ist und ohne großen Aufwand von fast jeder Programmiersprache gelesen werden kann.

Abgerundet wird das Framework durch die Möglichkeit, ausgehende Nachrichten verzögert zu senden, sowie falls nötig die Zeiten der von den Routern erzeugten Timer durch einen Faktor zu dehnen, der diese dadurch an die Verzögerung der ausgehenden Nachrichten anpasst. Auf diese Weise kann das Routing verlangsamt werden, sodass es besser beobachtet werden kann.

In den folgenden Abschnitten werde ich die Details der verschiedenen Packages des Frameworks vorstellen. Hierbei werden nur solche Funktionen, Klassen und Packages vorgestellt, die für Nutzer des Frameworks relevant sind. Frameworkinterne Details werden nur insoweit beleuchtet, wie sie für das Verständnis des Frameworks nötig sind. Abbildung 6.1 zeigt ein Klassendiagramm des Systems. Hier sind nur die wichtigsten Klassen und Uses-Beziehungen aufgeführt, da das Diagramm sonst zu unübersichtlich werden würde.

Alle Stellen des Frameworks nutzen Pythons flexibles Logger-Framework, um umfangreiche Logmeldungen sowohl auf der Konsole auszugeben, als auch via SSE an die Benutzerobefläche zu streamen. Die Konfiguration einzelner Klassen des Frameworks wird, sofern vorgesehen, über Einträge in der Settings-

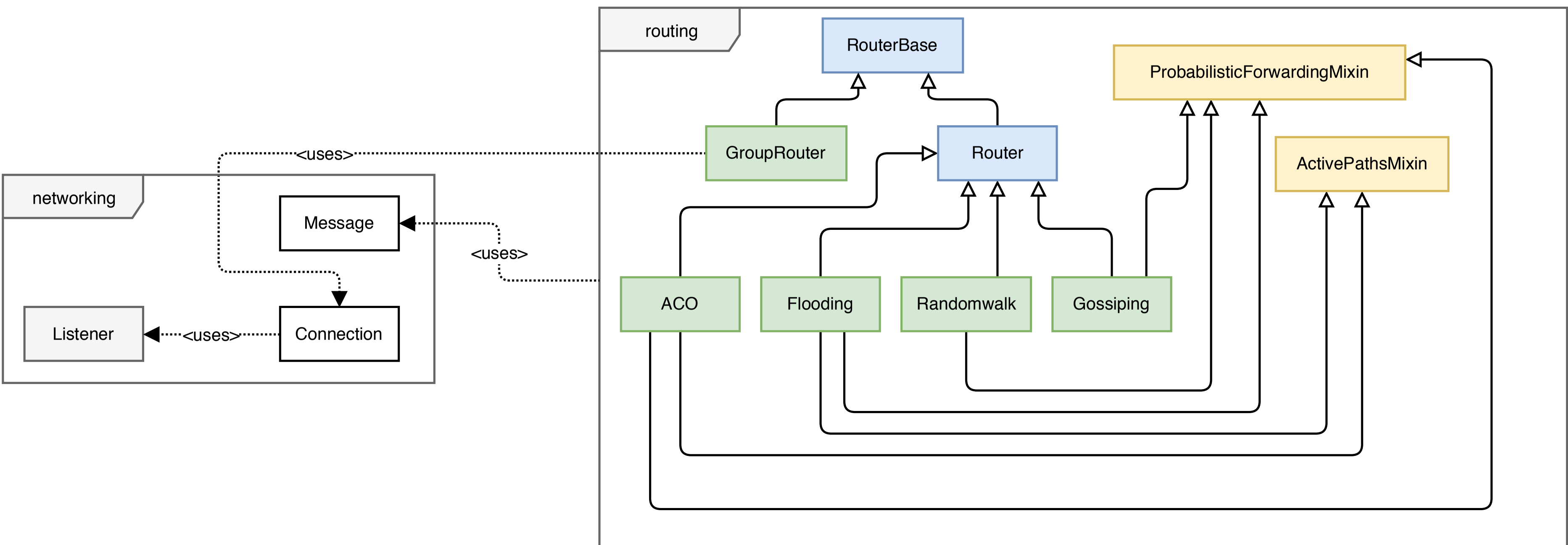


Abbildung 6.1.: Klassendiagramm des Systems. Routerklassen sind grün, die Basisklassen blau und die Mixins gelb. Graue Klassen werden von ihrem Package nicht exportiert.

Dictionary einer Klasse vorgenommen. In Abschnitt 6.2.4 wird beschrieben, wie diese Einstellungen zur Laufzeit eines Knotens verändert werden können.

6.2.1 Package *Networking*

Im Package *Networking* befinden sich alle für die Netzwerkfunktionalität relevanten Klassen.

Die Klasse *Connection* repräsentiert eine direkte Verbindung zwischen zwei Knoten über das, unter dem P2P Netzwerk liegende, TCP/IP Netzwerk, die Klasse *Listener* kümmert sich um das Empfangen von UDP [29] Paketen und die Klasse *Message* repräsentiert eine einzelne Nachricht, die zwischen den Knoten ausgetauscht werden kann.

Klasse *Listener*

Diese Klasse wird vom Package Networking nicht exportiert und ist nur für die Verwendung durch die Connection-Klasse vorgesehen. Der Listener kümmert sich um das Erzeugen eines UDP-Sockets, bindet diesen an einen Port (9999 für normale P2P Kommunikation, 9998 für die Kommunikation in Covergroups) und startet danach einen Thread, der auf den Empfang eines UDP-Paketes wartet.

Jedes empfangene UDP-Paket wird dann an die statische Methode **_incoming_data**() der Connection-Klasse weitergegeben, die als Factory Methode entweder eine neue Instanz der Klasse Connection erzeugt, oder eine für diesen Kommunikationsendpunkt bereits existierende Klasse verwendet, um dieser die empfangenen Daten zu übergeben.

So ist es möglich, auf einem verbindungslosen UDP-Port logische Verbindungen zu unterscheiden und diese in voneinander unabhängigen Instanzen der Klasse Connection zu verwalten.

Einstellungen der Klasse:
Diese Klasse besitzt keine Einstellungen.

Klasse *Message*

Die Klasse *Message* repräsentiert eine einzelne Nachricht, die zwischen den Knoten des P2P Netzwerks ausgetauscht werden kann.

Über die Methoden **set_type**() und **get_type**() kann der Typ der Nachricht gesetzt bzw. gelesen werden. Die Klasse implementiert darüber hinaus die drei magischen Methoden **__getitem__**(), **__setitem__**() und **__delitem__**(), sodass eine Instanz der Klasse syntaktisch genauso wie eine Dictionary genutzt werden kann. Werte, die über diese Zugriffsart gesetzt werden, landen in einer internen Dictionary der Klasse. Listing 6.1 zeigt die Verwendung der Klasse und den Inhalt der internen Dictionary.

Wird die Klasse zu einem String-Objekt umgewandelt, so wird der Inhalt der internen Dictionary zu einem String in JavaScript Object Notation (JSON)-Notation serialisiert. In der Message können daher nur die Python-Typen *dict*, *list*, *str*, *int*, *float* und *bool* abgelegt werden, wobei die verwendeten Keys stets Strings sein sollten.

```
msg = networking.Message("testmessage", {"abc": "xyz"})
msg["key_1"] = 42
msg["second_key"] = 24
# Die interne Repräsentation ist jetzt:
# {'type': 'test', 'data': {'key_1': 42, 'second_key': 24, 'abc': 'xyz'}}
```

Listing 6.1: Verwendung der Message-Klasse

Eine Umwandlung der Klasse in ein Byte-Objekt sorgt dafür, dass der JSON-String UTF-8-kodiert abgespeichert wird. Ein solcherart kodiertes Byte-Objekt lässt sich dann über das Netzwerk verschicken.

Wird der Konstruktor der Klasse mit einem String oder Byte-Objekt aufgerufen, so wird der darin enthaltene JSON String dekodiert und die Klasseninterne Dictionary mit den dekodierten Werten initialisiert. Werden dem Konstruktor keine Argumente übergeben, so ist die erzeugte Message leer. Dem Konstruktor können aber auch die beiden optionalen Argumente **msg_type** und **data** übergeben werden, die für die Initialisierung der Message verwendet werden.

Einstellungen der Klasse:
Diese Klasse besitzt keine Einstellungen.

Klasse *Connection*

Wie in Kapitel 4 bereits erwähnt, findet die Kommunikation über UDP [29] statt. Eine Verbindung zwischen zwei Knoten wird dabei durch die Klasse *Connection* repräsentiert. Diese Klasse kümmert sich um den Auf- und Abbau der Verbindung, sowie um das Senden der Heartbeat Messages und der darin versteckten Management Messages sowie um das Senden der normalen Datennachrichten.

Zur Simulation von zufälligen Paketverlusten auf der Netzwerkebene kann die Einstellung **PACKET_LOSS** verwendet werden, für die ein Wert im Bereich $(0, 1)$ erwartet wird. Dieser Wert gibt die Wahrscheinlichkeit an, mit der ein ausgehendes UDP-Paket *nicht* gesendet wird.

Im Folgenden werde ich die einzelnen Funktionen der Klasse erläutern und anschließend eine Übersicht über die öffentlich verwendbaren Methoden der Klasse geben.

Verschlüsselung:

Die Verbindung kann dabei mit Advanced Encryption Standard (AES) im Galois Counter Mode (GCM)-Modus verschlüsselt werden, wobei der hier verwendete Schlüssel über einen Diffie-Hellman-Schlüsselaustauch mit X25519 [32] stattfindet. Hierfür wird das Package *cryptography* mindestens in der Version 2.0 benötigt, da frühere Versionen X25519 nicht unterstützen. Außerdem muss auf manchen Plattformen zusätzlich *OpenSSL* in der Version 1.1 oder höher vorliegen, da *cryptography* diese Bibliothek intern verwendet. Sind diese Bedingungen nicht erfüllt, so muss die Verschlüsselung deaktiviert werden. Das ist beispielsweise auf Mac OS der Fall, da Apple von OpenSSL auf eine andere Bibliothek umgestiegen ist.

Verbindungsaufbau:

Da UDP verbindungslos ist, muss der Aufbau einer logischen Verbindung zwischen zwei Knoten durch die Connection-Klasse selbst übernommen werden. Eine Verbindung durchläuft dazu verschiedene Phasen, die an den Drei-Wege-Handshake von TCP [33] angelehnt sind. Abbildung 6.2 verdeutlicht den Handshake.

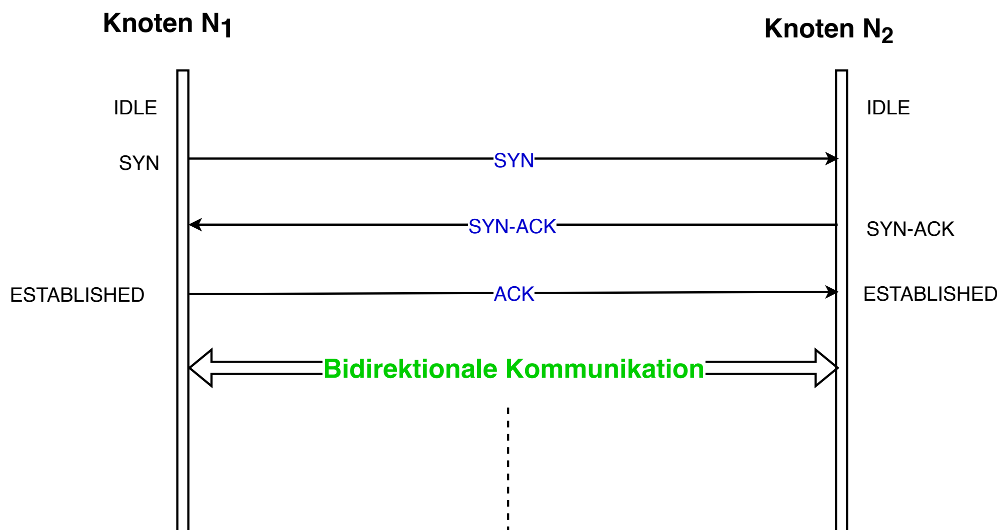


Abbildung 6.2.: Der Handshake der Connection-Klasse

Eine neu erzeugte Verbindung hat erst einmal den Status *IDLE*. Wurde die Verbindung durch den Aufruf von **connect_to**() erzeugt, so wird ein *SYN* gesendet und die Verbindung wechselt in den Status *SYN*. Wurde die Verbindung dagegen durch den Empfang eines UDP-Paketes vom Listener erzeugt, so wird auf den Empfang eines *SYN* gewartet und alle anderen UDP-Pakete ignoriert. Wird schließlich ein *SYN* empfangen, so wird ein *SYN-ACK* gesendet und die Verbindung wechselt in den Status *SYN-ACK*.

Ein Empfang von *SYN-ACK* im Status *SYN* löst das Senden eines *ACK* aus und die Verbindung wechselt in den Status *ESTABLISHED*. Gleiches gilt für den Empfang eines *ACK* im Status *SYN-ACK*. Alle anderen Übergänge sind nicht erlaubt und empfangene UDP-Pakete werden ignoriert. Eine Ausnahme hiervon ist der Empfang eines *SYN* im Status *SYN*. Hier wird ebenfalls ein *SYN-ACK* gesendet und in den Status *SYN-ACK* gewechselt.

Beim Verbindungsaufbau wird in den Nachrichten *SYN* und *SYN-ACK* außerdem die eigene Knoten-ID und ein öffentlicher X25519 Schlüssel für den Diffie-Hellman-Schlüsselaustausch mitgesendet. Der X25519 Schlüssel wird nur versendet, wenn die Verschlüsselung aktiviert wurde.

Verbindungsmanagement:

Ist die Verbindung aufgebaut, so wird ein Pinger-Thread gestartet, der für das, in der Einstellung **PING_INTERVAL**, angegebene Intervall (in Sekunden) eine Nachricht vom Typ **_ping** an den Verbindungspartner schickt. In dieser Nachricht werden die *Covert Messages* versteckt. Diese werden Base64-kodiert [34] in der Ping-Nachricht abgelegt. Dabei wird die konfigurierte Maximalgröße **MAX_COVERT_PAYLOAD** nicht überschritten. Das resultierende UDP-Paket kann allerdings größer werden, da die Ping-Nachricht selbst auch Platz beansprucht.

Jede Instanz der Klasse besitzt außerdem einen Watchdog-Thread, der bereits beim Erzeugen der Instanz gestartet wird. Wird dieser Watchdog nicht durch den Empfang von *Ping*-Messages regelmäßig zurückgesetzt, so wird die Verbindung nach **PING_INTERVAL** * **MAX_MISSING_PINGS**↩ Sekunden mittels **terminate**() beendet. Aktiv aufgebaute Verbindungen – das sind solche, die mittels **connect_to**() erzeugt wurden – werden dann nach einer zufälligen Zeitspanne der Größe **randint**(0,2)↩ + **PING_INTERVAL** * **MAX_MISSING_PINGS** wieder neu aufgebaut. Das erledigt ein eigens gestarteter Reconnect-Thread, der versucht nach Ablauf der Zeitspanne durch den Aufruf von **connect_to**() eine neue Verbindung aufzubauen.

Die minimale Wartezeit des Reconnect-Threads von **PING_INTERVAL** * **MAX_MISSING_PINGS** soll sicherstellen, dass die Verbindung beim Verbindungspartner ebenfalls schon als abgebrochen erkannt wurde, sodass der folgende Verbindungsaufbau reibungslos klappt. Ist die Verbindung bei der Gegenseite noch nicht als abgebrochen erkannt, so wird ein *SYN* vom Verbindungspartner verschluckt. Die zufällige Komponente der Wartezeit soll sicherstellen, dass bei einer zufälligen Kollision zweier Verbindungsversuche die beiden Kommunikationspartner nicht bei jedem erneuten Versuch eine Verbindung aufzubauen wieder eine Kollision erzeugen.

Die maximale Zahl an Verbindungsversuchen lässt sich über die Einstellung **MAX_RECONNECTS**↩ festlegen und ist standardmäßig auf drei gesetzt, während **PING_INTERVAL** auf 0,25 Sekunden und **MAX_MISSING_PINGS** in den Standardeinstellungen auf 16 festgelegt ist.

Einstellungen der Klasse:
Die Klassenvariable **settings** enthält eine Dictionary der in dieser Klasse verwendeten Einstellungen. Tabelle 6.1 zeigt die bereits in der Klasse gespeicherten Standardeinstellungen und erklärt diese. Veränderungen der Einstellungen müssen *vor* dem Aufruf der statischen Methode **init**() getätigt werden.

Einstellung Standardwert	**Beschreibung**
MAX_COVERT_PAYLOAD 1200	Größe des Payloads einer Ping-Nachricht. Die Zahl der gleichzeitig in einer Ping-Nachricht untergebrachten Covert Messages wird durch diese Einstellung beeinflusst. Die tatsächliche Payload-Größe des UDP-Pakets ist bei verschlüsselter Kommunikation noch um 94 Bytes größer, bei unverschlüsselter Kommunikation sind es 58 Bytes.
PING_INTERVAL 0.25	Intervall, in dem die Ping-Nachrichten ausgesendet werden.
MAX_MISSING_PINGS 16	Kommt diese Anzahl an Ping-Nachrichten hintereinander nicht an, so wird die Verbindung als abgebrochen betrachtet. Das Produkt **PING_INTERVAL** * **MAX_MISSING_PINGS** ergibt die Zeitspanne, die benötigt wird, um eine solche abgebrochene Verbindung zu erkennen. Diese Einstellung sollte so gewählt werden, dass diese Zeitspanne möglichst bei zwei bis vier Sekunden liegt.
MAX_RECONNECTS 3	Maximale Anzahl an neuen Verbindungsversuchen, bevor eine (abgebrochene) Verbindung endgültig als gescheitert betrachtet wird. Die Neuverbindungsversuche werden nur unternommen, wenn die Verbindung vorher aktiv durch einen Aufruf von **connect_to**() erzeugt wurde.
ENCRYPT_PACKETS **True**	Boolesche Einstellung, die angibt, ob die UDP-Pakete verschlüsselt übertragen werden sollen oder nicht.
PACKET_LOSS 0.0	Dieser Wert im Bereich $(0, 1)$ gibt an, wie groß die Wahrscheinlichkeit ist, dass ein UDP-Paket tatsächlich versendet wird und kann genutzt werden, um Netzwerkprobleme durch Paketverluste zu simulieren.

Tabelle 6.1.: Einstellungen der Klasse *Connection*

Methoden der Klasse:
Die Klasse bietet einige statische Methoden zur Erzeugung und Verwaltung von Verbindungen an, die in Tabelle 6.2 aufgelistet sind. Die klasseninternen statischen Methoden sind in dieser Tabelle nicht enthalten. Die öffentlichen Instanzmethoden der Klasse finden sich dagegen in Tabelle 6.3.

Statische Methode	Beschreibung
init()	Diese Methode muss vor der Verwendung der Klasse aufgerufen werden und initialisiert eine Instanz des Listeners, sowie die statischen Datenstrukturen der Klasse, die von den Factory-Methoden verwendet werden.
connect_to()	Hiermit kann eine neue Verbindung zu einem anderen Knoten aufgebaut werden.
disconnect_from()	Mit dieser Methode kann die Verbindung zu einem Knoten abgebaut werden. Besteht keine solche Verbindung, so tut diese Methode einfach nichts.
shutdown()	Diese Methode dient dem Beenden aller Verbindungen und stoppt den Listener. Sie ist als Gegenstück zur Methode **init**() zu verstehen. Nach dem Aufruf dieser Methode muss daher wieder die Methode **init**()↩ aufgerufen werden, um die Klasse neu zu initialisieren.

Tabelle 6.2.: Statische Methoden der Klasse *Connection*

Instanzmethode	Beschreibung
terminate()	Wird diese Methode aufgerufen, so wird die Verbindung beendet und alle dazugehörigen Threads beendet. Ein Neuverbinden findet nicht statt.
send_msg()	Mit dieser Methode kann eine einfache Datennachricht gesendet werden. Mit einem optionalen Parameter kann dafür gesorgt werden, dass der Aufruf des Filtersystems vor dem Versenden der Nachricht unterdrückt wird.
send_covert_msg()	Mit dieser Methode kann eine Nachricht im *Covert Channel* verschickt werden. Hierfür legt diese Methode die Nachricht in einer internen Queue ab. Beim nächsten Erzeugen einer Ping-Nachricht wird sie dann vom Pinger-Thread von dort abgeholt. Auch hier kann durch ein weiteres optionales Argument die Verwendung des Filtersystems für diese Nachricht deaktiviert werden.
get_peer_id()	Diese Methode liefert die Knoten-ID des Knotens am anderen Ende der Verbindung.
get_peer_ip()	Diese Methode liefert die IP-Adresse des Knotens am anderen Ende der Verbindung.
__repr__()	Diese magische Methode sorgt dafür, dass bei der Ausgabe eines Verbindungsobjektes in Logmeldungen ein formatierter Text mit der Knoten-ID, der IP-Adresse und einer Instanz-ID statt der Speicheradresse des Verbindungsobjektes ausgegeben wird.

Tabelle 6.3.: Instanzmethoden der Klasse *Connection*

Neben den Methoden **send_msg**() und **send_covert_msg**() bietet die Klasse eine Event-basierte Schnittstelle an, welche eingehende Nachrichten, neue Verbindungen oder auch Verbindungsabbrüche als Routerbefehle **add_connection** und **remove_connection** in die Router-Queue legt.

6.2.2 Package *Routing*

Im Package *Routing* befinden sich alle implementierten Router und deren, vom Framework angebotene, Basisklassen. Hierzu gehören auch einige Mixins, die den Routern häufig genutzte Funktionen wie das Management der aktiven Pfade zur Verfügung stellen.

Die Klasse *RouterBase* stellt die Basisklasse für alle Router dar. Darauf aufbauend stellt das Framework die Klasse *Router* zur Verfügung, die einige Pub/Sub-spezifische Funktionalitäten bereitstellt. Soll ein Router Pub/Sub implementieren, so muss er von der Klasse *Router* abgeleitet werden, wohingegen alle anderen Router *ohne* Pub/Sub-Semantik von der Klasse *RouterBase* abgeleitet sein müssen.

Die Mixins *ProbabilisticForwardingMixin* und *ActivePathsMixin* werden vom Framework bereitgestellt und können bei Implementierungen von Pub/Sub Routingalgorithmen als Basisklasse mit angegeben werden, um deren Funktionalität zu importieren. Mixin-Klassen müssen immer auf „Mixin“ enden, damit einige magische Funktionen des Frameworks funktionieren (siehe Abschnitt 6.2.3).

Soll ein Mixin konfigurierbar sein, so muss es die Methode **__configure**() implementieren, die vom Konstruktor des Routers aufgerufen werden kann. Jedes Mixin sollte darüber hinaus auch die Methode **__dump_state**() implementieren, die bei der Bearbeitung des Dump-Befehls genutzt wird, um den internen Status des Mixins auszugeben. Gibt es keinen internen Status, kann eine leere Dictionary zurückgegeben werden.

Es ist wichtig, dass alle Methodennamen eines Mixins (außer **__init__**()) mit einem doppelten Unterstrich beginnen. Dieses *DUnderscore* genannte Namensschema bewirkt, dass die Methode außerhalb der Klassendefinition um den Namen der Klasse erweitert wird. Dieses Verhalten nennt sich *Name Mangling* und sorgt dafür, dass die Methode nicht versehentlich von einer Kindklasse überschrieben wird (siehe [35][6]). Die Methode **__test**() eines Mixins namens *TestMixin* kann von der Kindklasse über den Namen **_TestMixin__test**() aufgerufen werden. Um Namenskonflikte auch bei Instanzvariablen zu vermeiden, müssen diese ebenfalls mit einem doppelten Unterstrich beginnen.

Klasse *RouterBase*

Diese Klasse ist die Basisklasse aller Router. In ihr ist der Router-Thread implementiert, der die von den Routern zu bearbeitenden Befehle aus der Router-Queue extrahiert und an die dafür im Router oder einem seiner Mixins vorgesehenen Methoden dispatcht.

Die in der Router-Queue gespeicherten Befehle sind Dictionaries, die der Struktur aus Listing 6.2 genügen müssen. Der Schlüssel **_command** muss zwingend angegeben werden, die restlichen Einträge der Dictionary enthalten frei wählbare optionale Argumente.

Wird der Befehl aufgerufen, so wird der Wert dieses Schlüssels zu _<**command_name**>**_command** erweitert und diese Methode auf der Routerklasse aufgerufen. Da Python Name-Mangling für Methoden mit vorangestelltem doppelten Unterstrich durchführt, muss für den Aufruf der Methode **__test_command** der Kindklasse *Testrouter* der Befehl **Testrouter__test** verwendet werden. Jede [...] **_command**-Methode muss ein einzelnes Argument erwarten, welches die Dictionary des Befehls enthält.

[6] Python Referenz: `https://docs.python.org/3/tutorial/classes.html#private-variables`

```
1 {
2     "_command": "test",
3     "data1": 42,
4     "some_more_data": "Testlauf",
5 }
```

Listing 6.2: Aufbau eines Router-Befehls

Diese Klasse sorgt außerdem dafür, dass eingehende Nachrichten anhand der Präfixe des Nachrichtentyps an die Kindklasse *oder* an eines ihrer Mixins weitergeleitet werden. Entspricht der Präfix des Nachrichtentyps den Namen der Kindklasse (beispielsweise **ACO**), so werden die Methoden **_route_covert_data**() bzw. **_route_data**() aufgerufen. Für Präfixe, die dem Namen eines Mixins genügen, werden dagegen die Methoden **__route_covert_data**() bzw. **__route_data**() des Mixins aufgerufen. Eine eingehende Management Message des Type **ACO_ant** wird beispielsweise an die Kindklasse **ACO** weitergeleitet, indem deren Methode **_route_covert_data**() aufgerufen wird.

Zur Bereitstellung dieser Funktionen sind die Befehlsmethoden **_covert_message_received_command**() und **_message_received_command**() implementiert, die auch dafür sorgen, dass das Filtersystem für eingehende Nachrichten aufgerufen wird. Sollen diese Methoden überschrieben werden, um die Verwendung des Filtersystems zu deaktivieren, kann die interne Methode **__route**() zum Dispatchen der eingehenden Nachrichten an die Mixins oder die Kindklasse verwendet werden.

Einstellungen der Klasse:
Diese Klasse besitzt keine Einstellungen.

Methoden der Klasse:
Die Klasse bietet einige öffentliche Methoden zur Steuerung eines Routers an, die in Tabelle 6.4 aufgelistet sind. Neben den öffentlichen Methoden stellt die Klasse ihren Kindklassen aber auch einige weitere Methoden zur Verfügung, die sich in Tabelle 6.5 aufgelistet finden.

Öffentliche Methode	Beschreibung
stop()	Mit dieser Methode kann ein Router gestoppt werden. Kindklassen müssen beim Überschreiben dieser Methode auch die Methode der Elternklasse aufrufen, damit der hier laufende Timer-Thread und Router-Thread beendet werden kann.
dump()	Mit dieser Methode kann eine Ausgabe der internen Datenstrukturen des Router erwirkt werden. Diese Methode legt einen entsprechenden Befehl in die Router-Queue.

Tabelle 6.4.: Öffentliche Methoden der Klasse *RouterBase*

Private Methode	Beschreibung
_add_timer()	Mit dieser Methode kann ein Timer erzeugt werden, der den übergebenen Router-Befehl zeitverzögert ausführt.
_abort_timer()	Diese Methode kann aufgerufen werden, wenn ein noch nicht abgelaufener Timer ohne Ausführung des Router-Befehls abgebrochen werden soll.
_call_command()	Mit dieser Methode kann ein Router-Befehl ausgeführt werden. Der Aufruf erfolgt im Gegensatz zu einem Timer mit dem Timeout 0 synchron. Das bedeutet, dass vor der Ausführung des hier angegebenen Befehls *keine* anderen Befehle ausgeführt werden.
_route_covert_data()	Diese Methode sollte von Kindklassen überschrieben werden und wird automatisch für eingehende Management Messages aufgerufen, die nicht für ein Mixin gedacht sind.
_route_data()	Diese Methode sollte ebenfalls von Kindklassen überschrieben werden und wird automatisch für eingehende Datennachrichten aufgerufen, die nicht für ein Mixin gedacht sind.
_add_connection_command()	Dieser Befehl ist bereits vorimplementiert und fügt neue Verbindungen der Instanzvariable **self.connections** hinzu. Sollen beim Hinzufügen besondere Aktionen ausgeführt werden, so kann diese Methode von Kindklassen überschrieben werden. Hierbei muss jedoch sichergestellt werden, dass auch die ursprüngliche Implementierung noch aufgerufen wird.
_remove_connection_command()	Auch dieser Befehl ist bereits Vorimplementiert und löscht die abgebrochenen Verbindungen aus **self.connections**. Sollen bei Verbindungsabbrüchen besondere Aktionen ausgeführt werden, kann diese Methode ebenfalls von Kindklassen überschrieben werden. Hierbei muss jedoch sichergestellt werden, dass auch die ursprüngliche Implementierung noch aufgerufen wird.

Tabelle 6.5.: Private Methoden der Klasse *RouterBase*

Klasse *Router*

Die Klasse *Router* baut auf der Klasse *RouterBase* auf und erweitert diese um einige Pub/Sub-spezifische Funktionen. Diese Klasse kann daher als Basis für Router verwendet werden, die Pub/Sub Routingalgorithmen implementieren.

Über die öffentlichen Methoden **subscribe**() und **unsubscribe**() bzw. **publish**() und **unpublish**() werden typische Pub/Sub-Funktionen bereitgestellt. Die Basisimplementierung der dabei ausgelösten Router-Befehle findet sich in den Methoden **_subscribe_command**(), **_unsubscribe_command**(), **_publish_command**() und **_unpublish_command**(). Die Implementierung in diesen Methoden fügt bei einer Subscription einen Eintrag in der Dictionary **self.subscriptions** hinzu, in welcher die Callbacks den Topics zugeordnet sind, die vom Knoten subscribed wurden. In der Instanzvariable **self.publishing** werden die Topics gespeichert, für die vom Knoten publiziert wird. Diese Variable ist als Set ausgeprägt (d.h. eine Liste die automatisch um Dopplungen bereinigt wird).

Kindklassen können diese Befehlsmethoden mit einer für ihren Routingalgorithmus spezifischen Implementierung überschreiben, die Methoden der Basisklassen sollten aber von der überschreibenden Methode weiter aufgerufen werden.

Einstellungen der Klasse:
Die Klassenvariable **settings** enthält eine Dictionary der in dieser Klasse verwendeten Einstellungen. Tabelle 6.6 zeigt die bereits in der Klasse gespeicherten Standardeinstellungen und erklärt diese.

Einstellung Standardwert	**Beschreibung**
OUTGOING_TIME 1.0	Dieser Wert kann verwendet werden, um ausgehende Nachrichten um diese Zeit in Sekunden zu verzögern. Das verbessert die Beobachtbarkeit des P2P Systems in Laborsituationen. Der Standardwert verzögert jede ausgehende Nachricht um eine Sekunde, simuliert also eine Datenübertragungsdauer des unter dem P2P liegende Netzwerks von einer Sekunde. Der Aufruf der Filter für ausgehende Nachrichten erfolgt *bevor* die Verzögerung angewendet wird.
TIMING_FACTOR 2.0	Über diesen Faktor lassen sich die in den Routern erzeugten Timer verlangsamen. Das sorgt dafür, dass sich die Timer der, durch **OUTGOING_TIME** ausgelösten, verzögerten Übermittlung von Nachrichten anpassen. Der Standardwert dehnt die Dauer jedes Timers auf das doppelte.

Tabelle 6.6.: Einstellungen der Klasse *Router*

Methoden der Klasse:
Ähnlich zur Klasse *RouterBase* bietet auch die Klasse *Router* einige öffentliche Methoden an, die spezifisch für die Steuerung eines Pub/Sub Routers sind und in Tabelle 6.7 aufgelistet werden. Neben den öffentlichen Methoden stellt die Klasse ihren Kindklassen aber auch einige weitere Methoden zur Verfügung, die sich in Tabelle 6.8 aufgelistet finden.

Öffentliche Methode	Beschreibung
subscribe()	Hiermit kann eine Subscription für ein Topic ausgelöst werden. Kommen Daten für dieses Topic an, wird der hier mit übergebene Callback ausgelöst.
unsubscribe()	Hiermit kann eine vorherige Subscription für ein Topic wieder rückgängig gemacht werden.
publish()	Hiermit können die übergebenen Daten zum übergebenen Topic veröffentlicht werden.
unpublish()	Sollen in Zukunft keine Daten mehr veröffentlicht werden, so kann diese Methode genutzt werden, um den Router über diesen Sachverhalt zu informieren.

Tabelle 6.7.: Öffentliche Methoden der Klasse *Router*

Private Methode	Beschreibung
_add_timer()	Hiermit wird die Methode der Basisklasse überschrieben, sodass neu angelegte Timer automatisch um die in den Einstellungen festgelegte Zeit gedehnt werden können.
_send_msg()	Diese Methode stellt eine Erweiterung der Methode **send_msg**() der Klasse *Connection* dar und sorgt dafür, dass ausgehende Nachrichten um die in den Einstellungen festgelegte Zeit verzögert gesendet werden. Kindklassen sollten diese Methode nutzen, anstatt direkt **send_msg**() auf einer Verbindung aufzurufen, um die Untersuchung ihres Verhaltens zu erleichtern.
_forward_data()	Diese Methode übergibt die empfangenen Daten eines Topics an den mit **subscribe**() registrierten Callback. Hierbei wird anhand einer in Datennachrichten immer mitgesendeten ID geprüft, ob die Daten schon einmal empfangen wurden. Ist das der Fall, so werden diese *nicht* an den Callback übergeben sondern ignoriert.
_add_connection_command()	Diese Methode erweitert die Funktion der Basisklasse um wichtige Funktionen, die die Zeitdehnung der Timer ermöglichen. Wird sie von einer Kindklasse überschrieben, muss sichergestellt werden, dass auch weiterhin die ursprüngliche Implementierung aufgerufen wird.
_remove_connection_command()	Auch hier werden wichtige Funktionen für die Zeitdehnung hinzugefügt, die die Basisklasse nicht bietet. Wird diese Methode von einer Kindklasse überschrieben, muss sichergestellt werden, dass auch weiterhin die ursprüngliche Implementierung aufgerufen wird.
_subscribe_command()	Diese Methode stellt eine Basisimplementierung der Subscribe-Funktionalität bereit.
_unsubscribe_command()	Diese Methode stellt eine Basisimplementierung der Unsubscribe-Funktionalität bereit.
_publish_command()	Diese Methode stellt eine Basisimplementierung der Publish-Funktionalität bereit.
_unpublish_command()	Diese Methode stellt eine Basisimplementierung der Unpublish-Funktionalität bereit.
_dump_command()	Diese Implementierung sollte von der Kindklasse *nicht* überschrieben werden. Die Kindklasse sollte stattdessen die Methode **__dump_state**() implementieren. Das stellt sicher, dass sowohl der interne Status der Kindklasse, als auch der ihrer Mixins erfasst werden kann, wenn ein Dump-Befehl verarbeitet wird.

Tabelle 6.8.: Private Methoden der Klasse *Router*

Klasse *ActivePathsMixin*

Dieses Mixin erweitert einen Pub/Sub Router um Funktionen zur Verwaltung der aktiven Kanten. Hierfür speichert es die vorwärts gerichteten Kanten, d.h. die Kanten vom Publisher zum Subscriber, in der internen Variablen **self.__active_edges** und die rückwäts gerichteten Kanten von einem Subscriber zum Publisher in der internen Variablen **self.__reverse_edges**.

Vom Mixin ausgesendete Nachrichten werden nicht unter dem Präfix des Mixins, sondern unter dem Präfix der Kindklasse versendet, um dieser die Möglichkeit zu geben, eine empfangene Nachricht noch

zu verändern, bevor diese an das Mixin weitergereicht wird. Diese Funktion wird beispielsweise von der Implementierung des Hashketten-basierten Routings verwendet.

Einstellungen der Klasse:
Die Methode **__configure(aggressive_teardown)** wird mit einem einzelnen boolschen Wert aufgerufen, der angibt, ob „herumhängende" Teile alter aktiver Pfade aggressiv abgebaut werden sollen, oder nicht (siehe hierzu die Erklärung in Abschnitt 4.5).

Methoden der Klasse:
Die Klasse bietet ihrer Kindklasse einige Methoden an, die in Tabelle 6.9 kurz erklärt werden. Durch das Name Mangling muss jeder Methodenname aus der Tabelle mit dem Präfix **_ActivePathsMixin**↩ versehen werden (beispielsweise **_ActivePathsMixin__configure()**. Dieses Präfix wurde in der Tabelle jedoch aus Platzgründen weggelassen.

Methode	Beschreibung
__configure()	Diese Methode wird zum Konfigurieren des Mixins verwendet und nimmt das boolsche Argument **aggressive_teardown**.
__init_channel()	Diese Methode kann von der Kindklasse aufgerufen werden, um die internen Datenstrukturen für ein Topic anzulegen. Falls nötig wird sie jedoch auch Mixin-intern aufgerufen.
__cleanup()	Diese Methode muss aufgerufen werden, wenn eine Verbindung abgebrochen ist und nimmt eine Instanz der Klasse *Connection* als Argument, die die abgebrochene Verbindung repräsentiert.
__send_error_reply()	Mit dieser Methode kann ein Router eine Error-Nachricht zurück an den Subscriber senden, wenn der Pfad nicht aktiviert werden kann, weil beispielsweie der nächste Schritt auf dem Pfad nicht gegangen werden kann. Das ist beispielsweise der Fall, wenn der Nachbarknoten, über den der aktive Pfad laufen soll nicht mehr verfügbar ist.
__unpublish()	Mit dieser Methode können alle aktiven Pfade eines Publishers abgebaut werden, wenn für diesen ein *Unpublish* ausgeführt wurde.
__get_next_hops()	Diese Methode liefert eine Liste von Instanzen der Klasse *Connection*, die zum Weiterleiten einer Datennachricht entlang der aktiven Pfade eines Topics genutzt werden müssen. Ein Router kann so ermitteln, an welche seiner Nachbarn eine Datennachricht weitergeleitet werden muss.
__active_edges_present()	Mit dieser Methode kann ein Router ermitteln, ob es einen aktiven Pfad zwischen einem Subscriber und einem oder allen Publishern eines Topics gibt. Sie gibt die Knoten-IDs aller Nachbarknoten zurück, für die ein solcher aktiver Pfad existiert.
__get_known_publishers()	Mit dieser Methode kann eine Liste aller bekannter Publisher-IDs ermittelt werden. Beim Hashketten-basierten Routing handelt es sich hierbei um Hashes, bei ACO sind es UUIDs [36].
__activate_edge()	Mit dieser Methode kann eine Kante von einem Nachbarknoten zu einem anderen aktiviert werden. Hierbei muss angegeben werden, zu welchem Subscriber und Publisher die Kante gehören soll und welche Version sie hat (siehe Abschnitt 4.5). Das Topic zu dem die Kante gehört und die zwei Instanzen der Klasse *Connection* werden ebenfalls benötigt.
__route_teardown()	Diese Methode muss für Teardown-Nachrichten aufgerufen werden und kümmert sich um deren Routing. Falls nötig kann die Message vom Router bearbeitet werden, bevor sie an diese Methode weitergegeben wird.
__route_error()	Diese Methode muss für Error-Nachrichten aufgerufen werden. Es gelten die gleichen Hinweise wie für **__route_teardown**(). Der optionale Parameter *callback* kann genutzt werden, um einen Callback zu hinterlegen, der aufgerufen wird, falls ein Neuaufbau des aktiven Pfades benötigt wird. Das ist generell nur der Fall, wenn ein Subscriber eine Error-Nachricht empfängt und keine aktive Kante mit einer höheren Versionsnummer als die der Error-Nachricht kennt.
__route_unsubscribe()	Diese Methode muss für Unsubscribe-Nachrichten aufgerufen werden. Es gelten die gleichen Hinweise wie für **__route_teardown**().
__route_unpublish()	Diese Methode muss für Unpublish-Nachrichten aufgerufen werden. Es gelten die gleichen Hinweise wie für **__route_teardown**().

Tabelle 6.9.: Methoden des Mixins *ActivePathsMixin*

Klasse *ProbabilisticForwardingMixin*

Dieses Mixin erweitert einen Pub/Sub Router um die Anonymisierungsfunktion PF (siehe Abschnitt 4.6.1).

Von diesem Mixin ausgesendete Nachrichten werden unter dem Präfix des Mixins versendet und daher unabhängig von der Kindklasse geroutet.

Einstellungen der Klasse:
Die Methode **__configure**(**probabilistic_forwarding_probability**) wird mit einem Float zwischen 0 und 1 aufgerufen, der die Wahrscheinlichkeit angibt, mit der ein Nachbarknoten für das PF ausgewählt wird.

Methoden der Klasse:
Die Klasse bietet ihrer Kindklasse einige Methoden an, die in Tabelle 6.10 kurz erklärt werden. Durch das Name Mangling muss jeder Methodenname aus dieser Tabelle mit dem Präfix **_ActivePathsMixin**↩ versehene werden (beispielsweise **_ActivePathsMixin__configure**(). Dieses Präfix wurde in der Tabelle jedoch aus Platzgründen weggelassen.

Methode	**Beschreibung**
__configure()	Diese Methode wird zum Konfigurieren des Mixins verwendet und nimmt dafür einen Float zwischen 0 und 1.
__add_pf_paths()	Diese Methode wird von einem Subscriber aufgerufen, wenn er sich durch PF vor einem Angreifer schützen möchte.
__get_additional_peers()	Diese Methode liefert dem Router eine Liste von Instanzen der Klasse *Connection*, die durch PF zum Attribute Overlay hinzugefügt wurden.

Tabelle 6.10.: Methoden des Mixins *ProbabilisticForwardingMixin*

6.2.3 Package *Utils*

Im Package *Utils* sind mehrere kleine nützliche Funktionen untergebracht, die ich hier kurz erläutern werde.

@final()
: Diese Klassenannotation sollte von den Routern verwendet werden und verhindert, dass diese als Basis für weitere Vererbung genutzt werden können.

@init_mixins()
: Diese Klassenannotation muss von Routern verwendet werden, die Mixins verwenden, damit diese korrekt initialisiert werden.

@catch_exceptions()
: Diese Annotation kann genutzt werden, um unbehandelte Exceptions eines Threads zu fangen und diese über den Logger auszugeben, bevor das Framework dann kontrolliert heruntergefahren und beendet wird.

get_class_that_defined_method()
: Diese Funktion ermittelt den Namen der Klasse, in der eine Methode definiert ist. Das ist nützlich,

um beispielsweise in einem Mixin das Präfix des Nachrichtentyps unabhängig von der genauen Namensgebung des Mixins zu halten.

init_leds()
: Mit dieser Funktion initialisiert das Framework die physischen und virtuellen LEDs der Benutzeroberfläche. Sie muss angepasst werden, falls sich die physische Ansteuerung der LEDs ändern sollte. Die LEDs können von einem geladenen Filter als Ausgabemöglichkeit verwendet werden (siehe Abschnitt 6.2.5).

6.2.4 Package *Control*

In diesem Package befindet sich die Klasse *Server*, die den http-Server bereitstellt, welcher auf Port 9980 lauscht und die REST- und SSE-Schnittstellen bereitstellt.

Unter dem Pfad „*/command*“ nimmt der Server JSON-kodierte http-POST-Anfragen entgegen, die er dekodiert als Python Dictionary in eine Command-Queue legt. Dort werden sie von der Mainloop entgegengenommen und bearbeitet.

Unter dem Pfad „*/events*“ kann mit einer http-GET-Anfrage der SSE-Stream abgerufen werden. Das Format dieses Streams wird in [37] beschrieben.

Unter dem Pfad „*/*“ oder „*/index.html*“ kann mit einem Browser das in Abschnitt 6.3 beschriebene Webbased Graphical User Interface (WebGUI) abgerufen werden. Im Unterordner *static* des Packages befinden sich die dazugehörigen statischen Dateien.

6.2.5 Filtersystem

Das Filtersystem des Frameworks ermöglicht es, über die REST-Schnittstelle beliebigen Pythoncode auf die Knoten des P2P Systems zu laden, selbst während der Router eines Knotens bereits läuft.

Diese Filter können dabei sowohl eingehende und ausgehende Nachrichten sehen, als auch die Router-Queue, die Command-Queue der Mainloop sowie die Event-Queue überwachen. Dazu muss der zu ladende Python-Code eine Klasse *Filters* enthalten, welche die in Tabelle 6.11 beschriebenen Methoden implementiert und von der Basisklasse *Base* abgeleitet sein muss. Ist der Rückgabewert einer Filtermethode `True`, so wird die gefilterte Nachricht oder der gefilterte Queue-Eintrag verworfen, ohne dass diese ihren Bestimmungsort erreichen.

In einer Instanz der Klasse *Filters* werden vom Framework außerdem die Variablen `self.leds` (LED-Steuerung), `self.router` (der aktuell laufende Router) und `self.group_router` (die Router-Implementierung eines ADC-net) verfügbar gemacht. Da die verschiedenen Methoden der Klasse von unterschiedlichen Threads aufgerufen werden, muss der Autor eines Filters selbst dafür sorgen, dass diese Aufrufe threadsicher verarbeitet werden. Insbesondere der parallele Zugriff auf Instanzvariablen muss korrekt implementiert oder ganz vermieden werden.

Listing 6.3 zeigt eine Beispielimplementierung eines Filters, der bei einer eingehenden *Covert Message* eine Logmeldung mit einer Zufallszahl ausgibt, bei einer eingehenden Datennachricht die 8. LED für 250 Millisekunden auf grün schaltet und ausgehende Datennachrichten grundsätzlich verwirft. Ein vollständiges Grundgerüst eines Filters kann in Listing A.1 gefunden werden.

Ein Filter muss nicht alle in Tabelle 6.11 aufgelisteten Methoden implementieren.

Methode	Beschreibung
__init__()	Sollte die Klasse einen Konstruktor haben, muss dieser das Argument **logger** konsumieren und an die Basisklasse weitergeben. Diese setzt dann die Instanzvariable **self.logger**. Soll die Konfiguration des Loggers verändert werden, so kann eine neue Instanz eines Loggers erzeugt werden und diese neue Instanz an die Basisklasse weitergegeben werden.
gui_command_incoming()	Diese Methode wird aufgerufen, sobald die Mainloop einen neuen Befehl aus der Command-Queue holt, aber bevor dieser verarbeitet wird.
gui_command_completed()	Diese Methode wird aufgerufen, wenn die Mainloop einen Befehl aus der Command-Queue bearbeitet hat. Ist dabei ein Fehler aufgetreten, so wird das optionale Argument **error** auf **True** gesetzt.
gui_event_outgoing()	Diese Methode wird aufgerufen, bevor ein Event aus der Event-Queue via SSE an eine Benutzeroberfläche gestreamt wird.
router_command_incoming()	Diese Methode wird aufgerufen, bevor der Router-Thread einen neuen Befehl aus der Router-Queue bearbeitet.
subscribed_datamsg_incoming()	Diese Filtermethode wird direkt vor dem Aufruf des bei der Subscription angegebenen Callbacks aufgerufen. Doppelt empfangene Datennachrichten sind an dieser Stelle im Gegensatz zur Filtermethode **msg_incoming**↩ () schon vom Framework herausgefiltert. Diese Methode bekommt die Datennachrichten daher genau so zu sehen, wie der Callback des Subscribers selbst.
covert_msg_incoming()	Diese Methode wird aufgerufen, wenn eine Nachricht auf dem *Covert Channel* empfangen wurde.
covert_msg_outgoing()	Diese Methode wird aufgerufen, bevor eine Nachricht auf dem *Covert Channel* versendet wird. Der Aufruf erfolgt *vor* einer etwaigen Verzögerung der Nachricht durch die Einstellung **OUTGOING_TIME** des Routers.
msg_incoming()	Diese Methode wird für eingehende Datennachrichten aufgerufen.
msg_outgoing()	Diese Methode wird für ausgehende Datennachrichten aufgerufen. Wie auch bei **covert_msg_outgoing()** bekommt der Filter die Nachricht zu sehen *bevor* sie durch den Router verzögert wird.

Tabelle 6.11.: Methoden eines Filters

```
1  class Filters(Base):
2      def __init__(self, logger):
3          # init parent class
4          super().__init__(logger)
5          # some imports needed later
6          self.random = __import__("random")
7
8      def covert_msg_incoming(self, msg, con):
9          self.logger.info("covert_msg_incoming: %d" % self.random.randint(0, 100))
10
11     def msg_incoming(self, msg, con):
12         self.leds[8].on((0, 255, 0), 0.25)
13
14     def msg_outgoing(self, msg, con):
15         return True
```

Listing 6.3: Beispiel eines Filters

6.2.6 Kommandozeilenoberfläche und Mainloop

Um einen Knoten zu starten, genügt es, die Datei *main.py* mit einem Python3 Interpreter zu starten. In diesem Fall lauscht der Knoten durch die Nutzung der Wildcard IP-Adresse *0.0.0.0* auf allen verfügbaren IP-Adressen. Soll der Knoten dagegen nur an einer spezifischen Adresse auf Verbindungen warten, so kann diese Adresse über das Kommandozeilenargument „*-l*“ angegeben werden.

Über das Kommandozeilenargument „*–log*“ kann darüber hinaus das Loglevel der Konsolenausgaben eingestellt werden. Das hier festgelegte Loglevel beeinflusst *nicht* das Streaming der Logmeldungen über SSE und kann zwischen *DEBUG*, *INFO*, *WARNING* und *ERROR* gewählt werden. Bei *–log WARNING* werden beispielsweise nur noch Meldungen der Stufe *WARNING* und *ERROR* ausgegeben.

Nach dem Start wartet ein Knoten auf Befehle über seine http-Schnittstelle. Startet man den Knoten ohne Argumente und navigiert danach mit einem Browser zu `http://127.0.0.1/`, so bekommt man die Web-basierte grafische Benutzeroberfläche des Knotens angezeigt (siehe Abschnitt 6.3). Listing 6.4 zeigt einen Start des Knotens ohne Kommandozeilenargumente.

```
$ ./main.py
2018-09-16 03:30:33,434 [INFO    ] root {MainThread} main.py:36: Logger ↩
    configured...
2018-09-16 03:30:33,435 [INFO    ] cherrypy.error {local::cherrypy_master} ↩
    _cplogging.py:214: [17/Sep/2018:03:30:33] ENGINE Bus STARTING
2018-09-16 03:30:33,435 [INFO    ] root {MainThread} main.py:64: My node id ↩
    is now 'ba3738dd-c79d-4b13-beb4-61d8354581fb'...
2018-09-16 03:30:33,436 [INFO    ] utils.led {MainThread} led.py:30: ↩
    Initializing LEDs...
2018-09-16 03:30:33,436 [INFO    ] cherrypy.error {local::cherrypy_master} ↩
    _cplogging.py:214: [17/Sep/2018:03:30:33] ENGINE Started monitor thread '↩
    _TimeoutMonitor'.
2018-09-16 03:30:33,543 [INFO    ] cherrypy.error {local::cherrypy_master} ↩
    _cplogging.py:214: [17/Sep/2018:03:30:33] ENGINE Serving on http↩
    ://0.0.0.0:9980
2018-09-16 03:30:33,543 [INFO    ] cherrypy.error {local::cherrypy_master} ↩
    _cplogging.py:214: [17/Sep/2018:03:30:33] ENGINE Bus STARTED
```

Listing 6.4: Ausgabe beim Start eines Knotens

Der Knoten behandelt nach dem Start in einer Mainloop, die im Hauptthread läuft, die über die REST-Schnittstelle ankommenden Befehle, die vom http-Server im Package *Control* in die Command-Queue gelegt werden.

6.3 Benutzeroberfläche

Die Benutzeroberfläche wurde mit Javascript geschrieben und kann im Browser ausgeführt werden. Zur Kommunikation mit den einzelnen Knoten wird die bereits beschriebene REST-Schnittstelle genutzt.

Die Benutzerschnittstelle erlaubt es, Graphen in dem von NetworkX erzeugten JSON-Format zu laden und zeigt diese animiert im Browserfesnter an. Für das automatische Layout des Graphen wird die

Javascript-Bibliothek *Springy* genutzt. Weitere genutzte Javascript Bibliotheken sind *JsPlumb*, *JQuery*, *JQuery-UI* und *LESS*. Alle verwendeten Bibliotheken stehen unter der MIT-Lizenz[7].

Alle Einstellungen der Klassen und die Auswahl des verwendeten Routers können komfortabel über eine Sidebar erledigt werden. Jede Einstellung besitzt einen Tooltip, der diese genauer erklärt und die Auswahlboxen sind so eingestellt, dass unsinnige Werte nicht eingestellt werden können.

Läd man über den entsprechenden Button eine Filterdatei auf die angezeigten Knoten, so kann man die vom Filter angesteuerten LEDs als kleine Punkte auf jedem Knoten „leuchten" sehen, sofern das Routing auf den Knoten durch den Button *Start Routers* gestartet wurde.

Die Oberfläche erlaubt, es zur Laufzeit des P2P Netzwerks mit der Maus neue Verbindungen zwischen Knoten zu erzeugen oder diese zu beenden. Klickt man auf einen Knoten, so bekommt man die Detailansicht dieses Knotens präsentiert. Hier kann man – ebenfalls auch zur Laufzeit – neue Subscriptions erzeugen oder Subscriptions beenden, sowie einen Knoten für ein Topic publizieren lassen.

Das ebenfalls angezeigte Logfenster zeigt die Logausgabe des gerade ausgewählten Knotens in Echtzeit an und kann genutzt werden, um einen genauen Blick auf die stattfindenden Abläufe zu werfen. Für diesen Zweck kann darüber hinaus auch der Button *Dump internal state of router* genutzt werden, welcher den Router und seine Mixins veranlasst, seinen Status als JSON-kodierten String in dem dafür vorgesehenen Textfeld auszugeben.

Die grafische Oberfläche ist vollkommen vom restlichen System entkoppelt und es ist auch möglich die Oberfläche zu beenden, während die Router weiter laufen. Durch die Entkopplung sind ebenfalls alternative Oberflächen für die Steuerung der Knoten denkbar. Eine weitere solche Oberfläche stelle ich in Abschnitt 7.1 vor. Hier wird die REST-Schnittstelle und das Filtersystem dazu genutzt, um vollautomatisch verschiedene Messungen an einem 100 Knoten großen P2P Netzwerk durchzuführen.

Die Abbildungen 6.3 und 6.4 zeigen die Oberfläche und das System in Aktion.

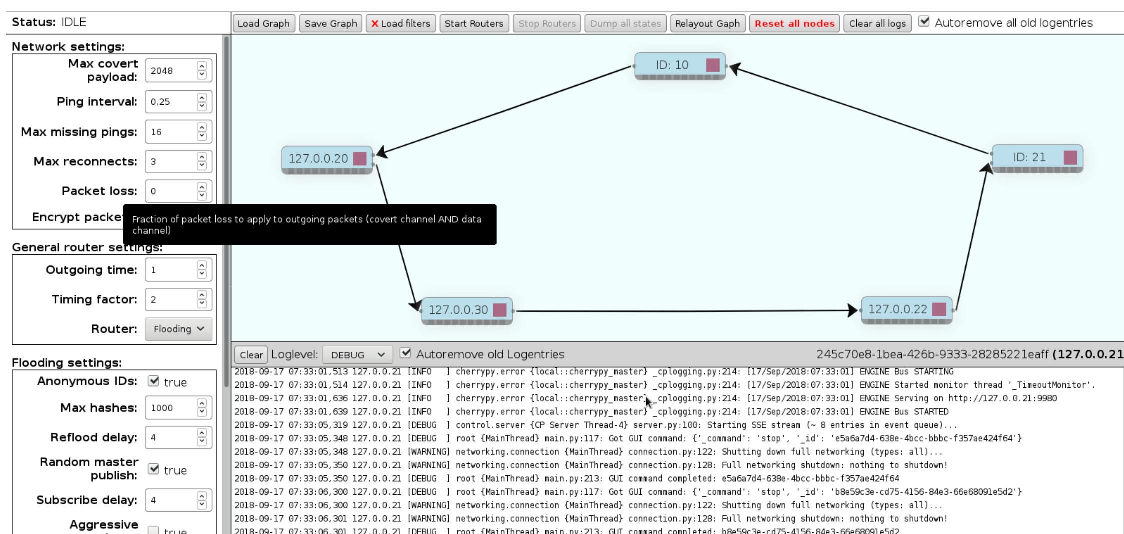


Abbildung 6.3.: Die Einstellungen in der Sidebar

7 `https://opensource.org/licenses/MIT`

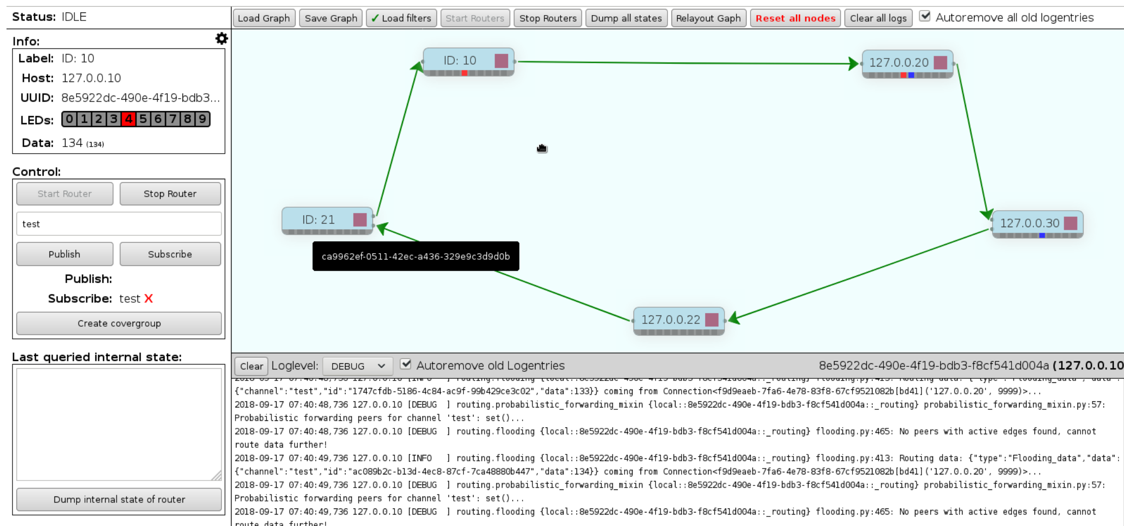


Abbildung 6.4.: Das laufende System mit der Detailansicht eines Knotens in der Sidebar

6.4 Implementierung der Router

Die einzelnen Router sind in den Klassen *ACO* (ACO) und *Flooding* (Hashketten-basiertes Routing) implementiert. Die Klassen *Randomwalk* und *Gossiping* implementieren ein einfaches Randomwalk- bzw. Gossiping-basiertes Routing ohne den Aufbau eines Attribute Overlays. Der vollständige Abdruck des nur etwa 50 Zeilen langen Randomwalk-Routers kann in Listing A.2 gefunden werden.

Alle Router verwenden das bereits beschriebene Framework, um die in Kapitel 4 und 5 erarbeiteten und ausführlich beschriebenen Konzepte und Lösungen umzusetzen.

Die Klasse *GroupRouter* implementiert die Kommunikation in ADC-net Covergroups. Da es sich bei Covergroups um kein Pub/Sub-Konzept handelt, ist dieser Router von der Klasse *RouterBase* abgeleitet und nicht von der Klasse *Router*.

7 Evaluation

In Kapitel 4 und 6 werden einige Parameter des Hashketten-basierten und des ACO Routings erwähnt und Standardwerte vorgeschlagen. Einige dieser Werte sollen nun in einer Evaluation getestet werden.

Für jede Simulation wurden hierfür auf einem Rechner 100 Knoten unter den IP-Adressen *127.0.0.100* bis *127.0.0.199* gestartet, die in einem zufälligen Graphen mit Durchschnittsgrad vier verbunden waren. Von diesen Knoten wurden drei Publisher und zehn Subscriber zufällig ausgewählt, die alle ein einziges Attribute Overlay für den Channel „test“ aufgebaut haben.

Die Netzwerkverzögerung wurde auf 20 Millisekunden und das Intervall der Heartbeat Messages auf 100 Millisekunden eingestellt. Eine Management Message kann ohne Paketverlust auf Netzwerkebene und bei einem nicht durch andere Management Messages verstopften *Covert Channel* maximal 120 Millisekunden pro Knoten verzögert werden. Nicht eingerechnet sind hierbei die Verzögerungen, die durch die Bearbeitung einer Management Message auf den Knoten selbst entstehen. Als Größe der Management Messages wurden 16 Kibibyte gewählt.

Die vollständigen Einstellungen der Knoten, die in allen Messungen verwendet wurden, findet sich in Listing A.5. Wurde von diesen Einstellungen abgewichen, ist das entsprechend notiert.

Damit zufällige Schwankungen die Messungen nicht beeinflussen, wurden alle Messungen 15 Mal durchgeführt und der Durchschnittswert dieser 15 Messungen ermittelt (das gilt auch für eventuell gemessene Minima und Maxima). Da in dieser Evaluation das Echtzeitverhalten des Systems gemessen werden soll, wurden alle Messungen außerdem *ohne* ein verzögertes Senden von Nachrichten oder eine Verlangsamung der Timer durchgeführt.

7.1 Framework

Wie in Kapitel 6 bereits geschildert, besitzt jeder Knoten eine REST-basierte Schnittstelle, über die er gesteuert werden kann. Das Evaluationsframework nutzt diese Schnittstelle zusammen mit dem Filtersystem jedes Knotens (siehe Abschnitt 6.2.5), um vollautomatische Messungen vorzunehmen. Alle Logausgaben und der verwendete Graph mit allen Einstellungen wird in einem Ordner archiviert, dessen Name sich aus dem Namen des Tasks (siehe Abschnitt 7.1.2) und der Nummer der Messung sowie der Nummer der Iteration über die Einstellungswerte zuammensetzt. Dieser Ordner kann später verwendet werden, um den Testlauf vollständig zu rekonstruieren.

7.1.1 Filter

Die Filter, die die LED-Ausgabe steuern, können auch genutzt werden, um Logmeldungen auszugeben. Da die Filter fast alle Interna eines Knotens sehen können, lassen sich damit Messungen an einem Knoten durchführen, die dann als Logmeldung ausgegeben werden können.

Die zu messenden Werte werden im Filter ermittelt und über eine spezielle Logmeldung ausgegeben, die dem Schema „************ **CODE_EVENT**(<**eventname**>): <**code**>“ folgt, wobei der ausgegebene Code

beliebiger *einzeiliger* Python-Code sein kann und der Eventname ein beliebiger Name sein kann. Dieser Name wird nicht verwendet und dient nur der besseren Identifikation in einer Logdatei. Die Logmeldungen *aller* Knoten werden in ihrer korrekten zeitlichen Abfolge in einer einzelne Logdatei zusammengefasst und die darin enthaltenen Codezeilen in dieser Reihenfolge ausgeführt.

Dem Code in der Filterdatei steht die global gebundene Variable *task* zur Verfügung, die die komplette Beschreibung des Tasks aus dem Taskfile als verschachtelte Python Dictionary enthält. Hiermit können Entscheidungen auf Basis des gerade ausgeführten Tasks getroffen werden.

7.1.2 Taskfile

Verschiedene Messungen benötigen oftmals unterschiedliche Parameter und die rohen Werte aus der Auswertung der Logdatei-Codezeilen müssen meist noch weiter aufbereitet werden. Hierfür wird eine umfangreiche Konfigurationsmöglichkeit benötigt, die ich auf Basis einer JSON-Datei implementiert habe. Ein vollständig kommentiertes Minimalbeispiel einer solchen Datei findet sich in Listing A.3. Die Ergebnisse der Messungen werden ebenfalls im JSON-Format ausgegeben (siehe Listing A.4).

Dem in dieser Datei an einigen Stellen angegebenen Pythoncode stehen, neben den Built-in Funktionen von Python, die Packages *numpy* und *random* sowie die Funktion *reduce* aus dem Package *functools* als lokal gebundene Variablen zur Verfügung.

Hinweis zur Notation: Ein Punkt im Namen einer Einstellung bedeutet in der JSON-Notation aller Einstellungen einen Abstieg auf die nächste Ebene der verschachtelten Objekte, aus denen die Einstellungen zusammengesetzt sind. Listing 7.1 zeigt die Darstellung der Einstellung *routing.ACO.ANT_ROUND_TIME* in JSON-Notation.

```
1 {
2     "routing": {
3         "ACO": {
4             "ANT_ROUND_TIME": 5
5         }
6     }
7 }
```

Listing 7.1: Die Einstellung *routing.ACO.ANT_ROUND_TIME* in JSON-Notation

Optionales Feld *iterate*

Das optionale Feld *iterate* nimmt die zwei Felder *setting* und *iterator* auf. In *setting* steht hierbei der Name der zu verändernden Einstellung in Punktnotation (beispielsweise **routing.Flooding↩ .SUBSCRIBE_DELAY**) oder ein Javascript-Array, der die zu verändernden Einstellungen auflistet (ebenfalls in Punktnotation).

Das Feld *iterator* ist ein String, der einen möglichst einzeiligen Pythoncode aufnimmt, welcher ausgeführt ein iterierbares Objekt zurückgibt (beispielsweise *[1, 2, 4]*). Wurden in *setting* mehrere Einstellungen aufgelistet, muss der Iterator für jede Iteration ein Tupel zurückgeben, welches für jede der genannten Einstellungen einen Wert enthält. Die Zuordnungsreihenfolge ist hierbei von links nach rechts gegeben.

Feld *init*

Dieses Feld enthält die Initialisierungswerte des vom Filter generierten Codes als Javascript-Objekt mit dem Variablennamen als Key.

Feld *output*

Dieses Feld dient der Überführung der Rohvariablen aus dem Filtercode (d.h. der Variablen aus dem Feld *init*) in neue Werte. Hierbei können beispielsweise im Filtercode generierte Listen zu einem einzelnen Wert akkumuliert werden. Hierfür wird der Name der dabei neu zu generierenden Variablen als Key und der (möglichst einzeilige) Pythoncode, der zur Generierung dieser Variablen genutzt wird, als Value eines Javascript-Objekts abgelegt. Ist das Feld *output* nicht vorhanden, so gilt der Task als deaktiviert.

Optionales Feld *reduce*

Dieses besitzt den gleichen Aufbau wie das Feld *output* und kann dazu genutzt werden die Ergebnisse einzelner Simulationsrunden zu einem neuen Wert zu akkumulieren. Dem (möglichst einzeiligen) Pythoncode stehen dafür die zu bearbeitenden Ergebnisse in der Variablen *valuelist* zur Verfügung. Werden einzelne Variablen oder auch das ganze Feld weggelassen, so findet für die betroffenen Variablen keine Akkumulierung der Rundenergebnisse statt.

7.2 Hashketten-basiertes Routing

In diesem Abschnitt werde ich die verschiedenen Messungen vorstellen, die am Hashketten-basierten Routing durchgeführt wurden.

7.2.1 Dauer des Attribute Overlay Aufbaus

Für diese Evaluation wird ein Overlay als aufgebaut betrachtet, sobald das erste Mal Datennachrichten von allen Publishern bei allen Subscribern eintreffen.

Die verwendeten Einstellungen:
$routing.Flooding.MIN_BECOME_MASTER_DELAY = 4$
$routing.Flooding.MAX_BECOME_MASTER_DELAY = 20$
$routing.Flooding.SUBSCRIBE_DELAY = 3$

Die Dauer einer einzelnen Messung beträgt 45 Sekunden. Abbildung 7.1 zeigt das Ergebnis dieser Evaluation. Der Aufbau des Attribute Overlays dauert durchschnittlich $13,1$ Sekunden, mit einem Minimum von $11,4$ Sekunden und einem Maximum von $14,65$ Sekunden.

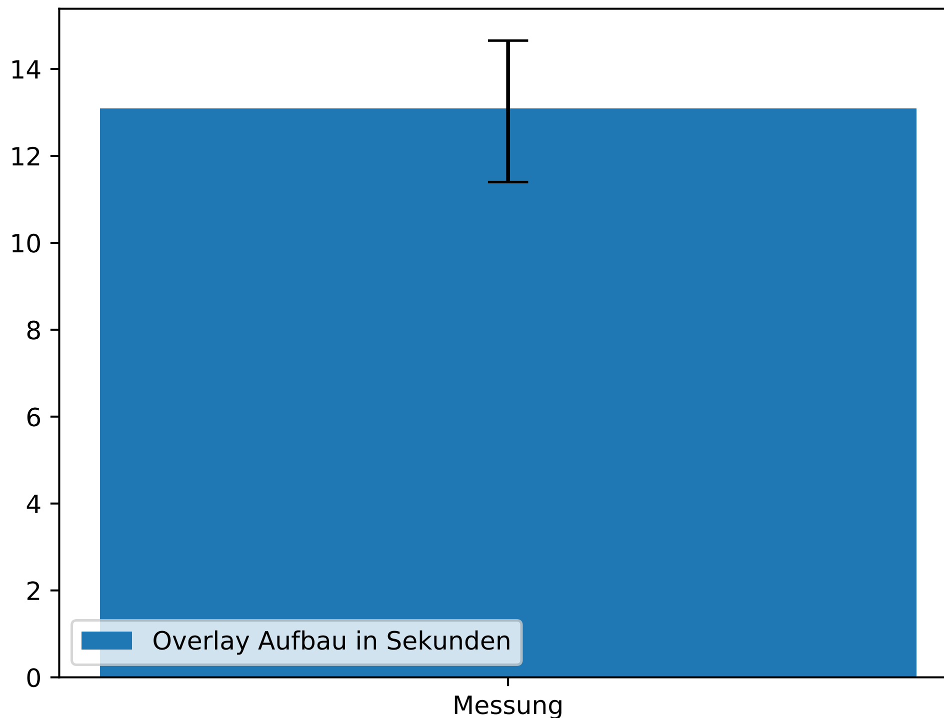


Abbildung 7.1.: Dauer des Attribute Overlay Aufbaus bei Hashketten-basiertem Routing

7.2.2 Suboptimale Pfade

Ist das Fluten der Advertisements abgeschlossen, so kennt jeder Knoten den nächsten Schritt auf dem kürzesten Weg zu allen Mastern. Ist das Fluten jedoch noch nicht vollständig abgeschlossen, dann kann es unter Umständen sein, dass ein Knoten zwar einen Pfad zum Master kennt, aber dieser Pfad nicht der optimale ist. Wird ein Pfad aktiviert, wenn die involvierten Knoten noch nicht alle den optimalen Weg kennen, dann ist dieser Pfad zwar funktionstüchtig, aber nicht optimal (siehe Abschnitt 5.1.5).

In dieser Evaluation wird die Anzahl dieser suboptimalen Pfade gemessen. Hierbei wird zwischen solchen Pfaden, deren Weg an einem Forwarder vom optimalen Pfad abzweigt, und solchen, bei denen die Abzweigung direkt am Subscriber vorliegt, unterschieden.

Lässt man die Subscriber einige Sekunden warten, bevor sie einen Pfad aktivieren, kann man die Wahrscheinlichkeit erhöhen, dass alle Advertisements vollständig geflutet wurden. Wie lange hierfür gewartet werden soll, wird in dieser Evaluation ermittelt, indem die Einstellung $routing.Flooding.SUBSCRIBE_DELAY$ im Wertebereich $(0,5)$ mit Schrittweite $0,5$ evaluiert wird. Die Dauer einer einzelnen Messung beträgt 45 Sekunden.

Abbildung 7.2 zeigt das Ergebnis dieser Evaluation. Hatte ein einzelner Subscriber zu mehreren verschiedenen Master-Publishern suboptimale Pfade, so wurden diese ebenfalls zur Gesamtsumme aufaddiert, so wie die suboptimalen Pfade verschiedener Subscriber auch.

Die Zahl der suboptimalen Pfade an den Subscribern fällt nach einem anfänglichen Hoch von durchschnittlich $1,3$ suboptimalen Pfaden bei keiner Wartezeit, auf Null suboptimale Pfade bei einer Wartezeit von zwei Sekunden. Die suboptimalen Pfade an Forwardern fällt sogar schon bei einer Wartezeit von

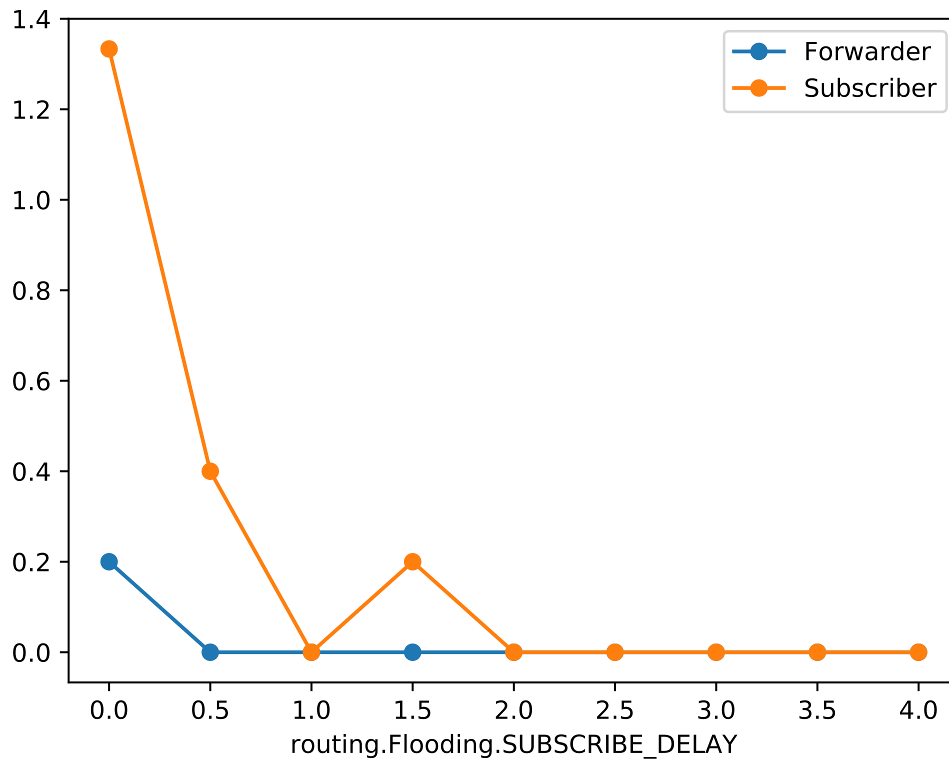


Abbildung 7.2.: Anzahl der suboptimalen Pfade bei Hashketten-basiertem Routing mit drei Publishern

0, 5 Sekunden auf Null, nachdem anfänglich durchschnittlich 0, 4 suboptimale Pfade an den Forwardern abzweigen.

Verwendet man dagegen fünf Publisher, ergibt sich das in Abbildung 7.3 gezeigte Bild. Hier fällt der Anteil der suboptimalen Pfade erst bei einer Wartezeit von drei Sekunden dauerhaft auf Null. Sowohl die durchschnittliche Zahl der suboptimalen Pfade an den Subscribern (1, 9) als auch die Zahl dieser Pfade an den Forwardern (0, 3) hat bei 0, 5 Sekunden ihren Höhepunkt.

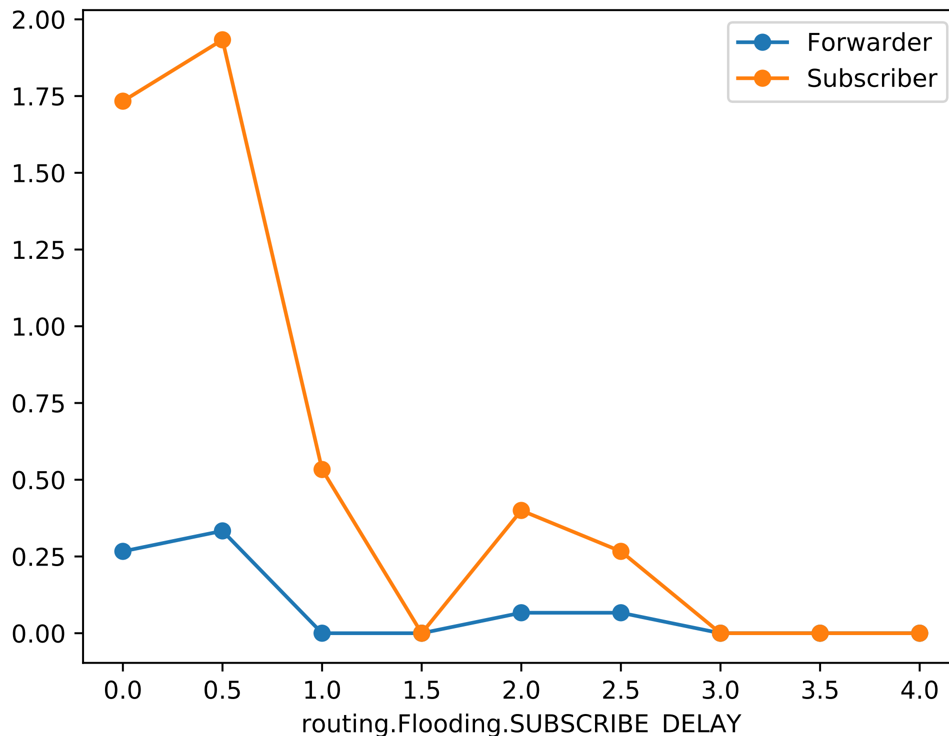


Abbildung 7.3.: Anzahl der suboptimalen Pfade bei Hashketten-basiertem Routing mit fünf Publishern

7.2.3 Master-Publisher

Wie in Abschnitt 5.1.3 geschildert, können beim initialen Verbinden eines P2P Overlays etwa zeitgleich mehrere Master-Publisher entstehen, da sich die Information über die anderen Publisher noch nicht über das Netzwerk ausgebreitet hat.

Als Gegenmaßnahme kann man einen angehenden Publisher eine zufällig gewählte Zeitspanne warten lassen, bevor dieser sich entscheidet, ob er Master oder Slave wird. Sind in dieser Zeit Advertisements eines anderen Publishers angekommen, so wird der angehende Publisher ein Slave-Publisher, ansonsten wird er ein Master-Publisher und flutet den Overlay selbst mit Advertisements. Die beiden Einstellungen *routing.Flooding.MIN_BECOME_MASTER_DELAY* und *routing.Flooding.MAX_BECOME_MASTER_DELAY* geben hierfür den Korridor vor, in dem die Wartezeit zufällig gewählt werden kann.

In dieser Evaluation soll die Anzahl der Master- und Slave-Publisher in Abhängigkeit der möglichen Wartezeit gemessen werden. Hierfür wird *routing.Flooding.MIN_BECOME_MASTER_DELAY* fest auf den Wert 2 gesetzt, während *routing.Flooding.MAX_BECOME_MASTER_DELAY* im Bereich $(10, 20)$ mit Schrittweite 1 verändert wird, wodurch sich der mögliche „Wartekorridor“ vergrößert. Die Dauer einer einzelnen Messung beträgt 35 Sekunden.

Abbildung 7.4 zeigt das Ergebnis dieser Evaluation.

Wie zu sehen ist, bewegt sich die Anzahl der Master-Publisher zwischen $1,4$ (47%) und $1,9$ (63%). Mit einer Anzahl von fünf Publishern ergibt sich das in Abbildung 7.5 gezeigte Bild. Hier bewegt sich die Zahl der Master-Publisher zwischen $1,6$ (32%) und 3 (60%).

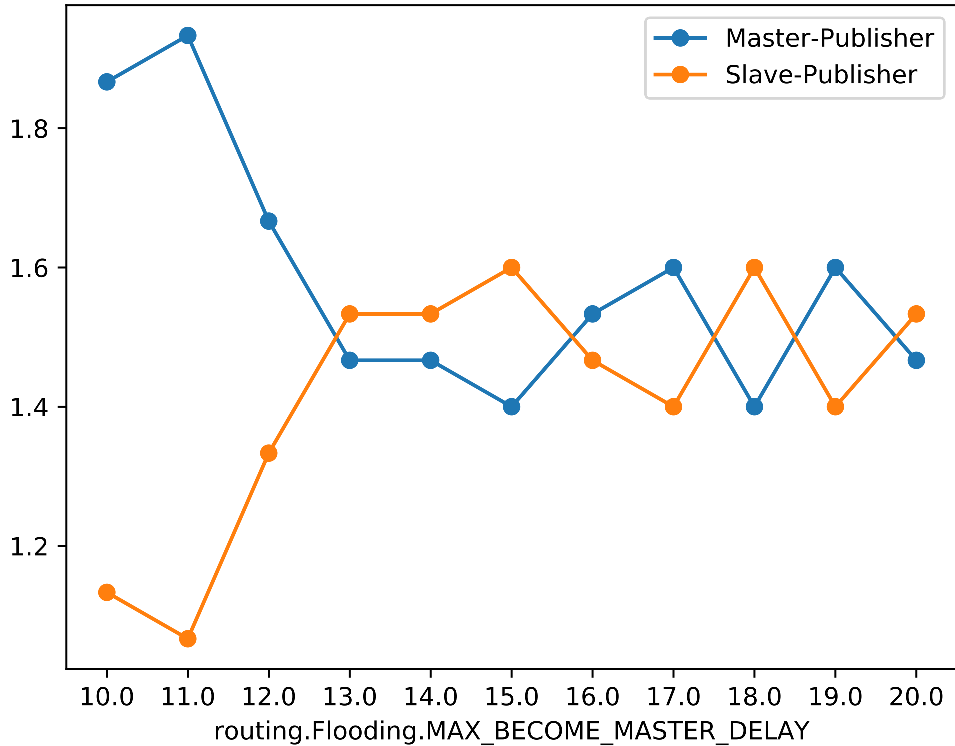


Abbildung 7.4.: Anzahl der Master und Slaves bei Hashketten-basiertem Routing mit drei Publishern

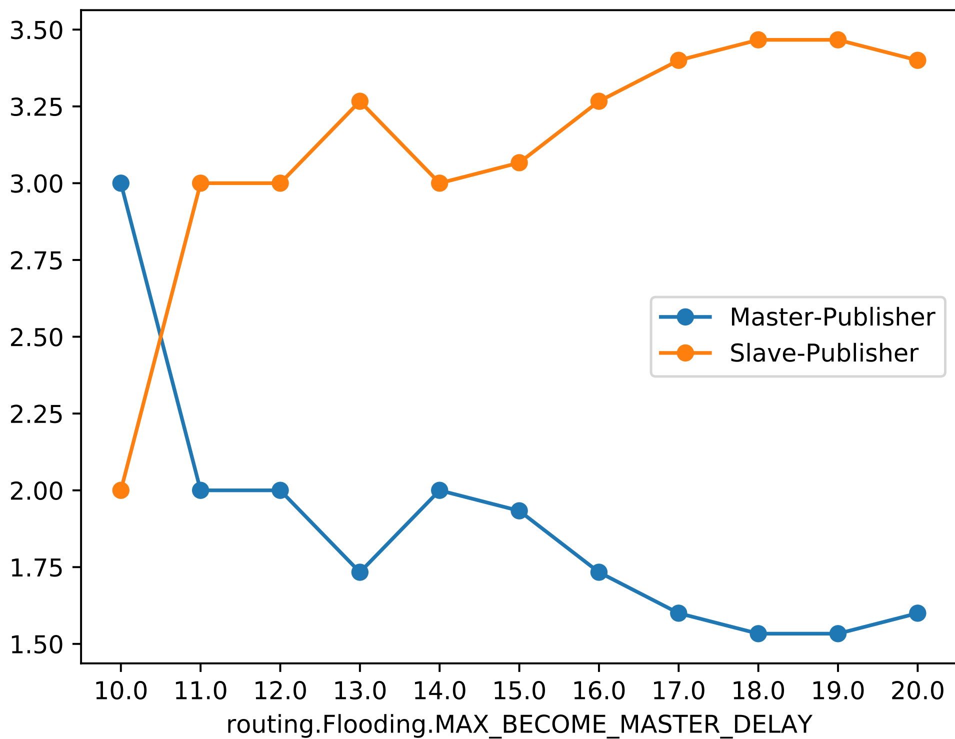


Abbildung 7.5.: Anzahl der Master und Slaves bei Hashketten-basiertem Routing mit fünf Publishern

7.3 ACO Routing

In diesem Abschnitt werde ich die verschiedenen Messungen vorstellen, die am ACO-basierten Routing durchgeführt wurden.

Die hierbei verwendeten Basiseinstellungen sind:
$routing.ACO.ANT_COUNT = 10$
$routing.ACO.EVAPORATION_TIME = 4$
$routing.ACO.EVAPORATION_FACTOR = 0.75$
$routing.ACO.DEFAULT_ROUNDS = 100$
$routing.ACO.ACTIVATION_ROUNDS = 5$
$routing.ACO.ACTIVATING_ANTS = 2$
$routing.ACO.ANT_ROUND_TIME = 2$

7.3.1 Dauer des Attribute Overlay Aufbaus

Für diese Evaluation wird ein Overlay als aufgebaut betrachtet, sobald das erste Mal Datennachrichten von allen Publishern bei allen Subscribern eintreffen.

Die Einstellung $routing.ACO.ANT_ROUND_TIME$ wurde hierfür im Wertebereich $(1, 4)$ mit Schrittweite $0,5$ verändert. Die Zahl der Ameisen wurde auf 10, der Verdampfungsfaktor auf $0,75$, das Intervall der Verdampfungsrunden auf 4 Sekunden und die maximale Zahl der Ameisenrunden auf 100 eingestellt. Die Dauer einer einzelnen Messung beträgt 120 Sekunden.

Abbildung 7.6 zeigt das Ergebnis dieser Evaluation. Nicht immer konnte dabei der Attribute Overlay komplett aufgebaut werden. In diesem Fall wurden nur die Werte der 15 Messungen berücksichtigt, bei denen alle 10 Subscriber vollständig verbunden waren. Abbildung 7.7 zeigt die Anzahl dieser vollständig verbundenen Subscriber.

Es ist zu erkennen, dass die für den Aufbau des Attribute Overlays benötigte Zeit mit der Rundenzeit von anfänglich durchschnittlich $14,5$ auf später $37,6$ Sekunden steigt. Gleichzeitig schwankt allerdings die Zahl der erfolgreich verbundenen Subscribern stark und erreicht zwischenzeitlich sogar in einer der 15 gemittelten Messungen Null. Durchschnittlich schaffen es allerdings zwischen $9,8$ und 10 Subscribern, sich erfolgreich mit allen Publishern zu verbinden.

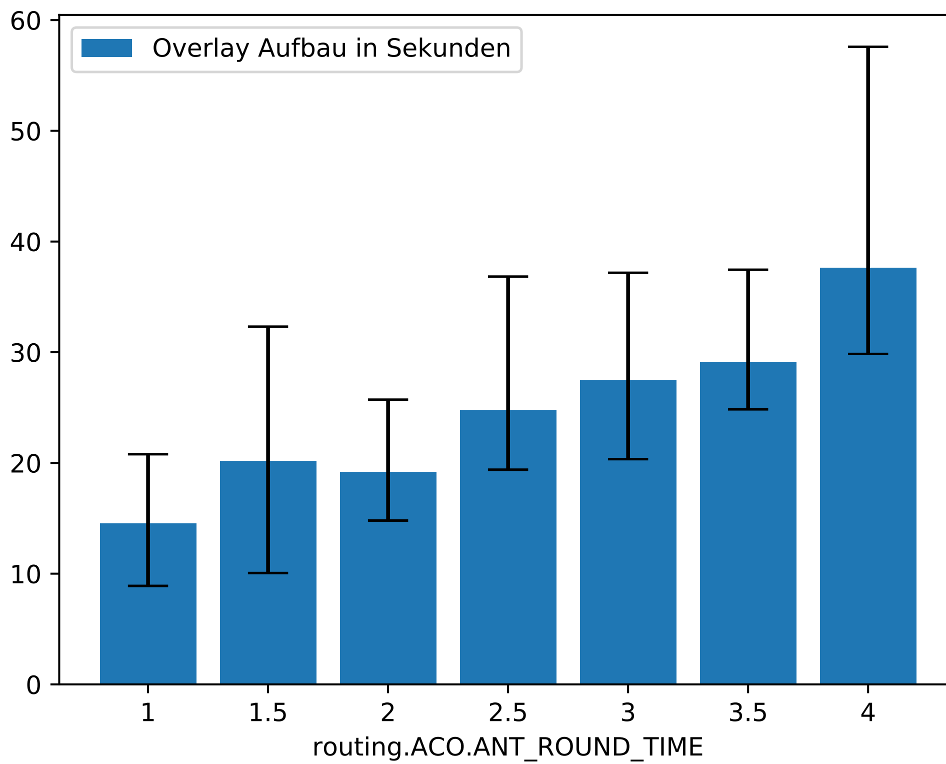


Abbildung 7.6.: Dauer des Attribute Overlay Aufbaus bei ACO

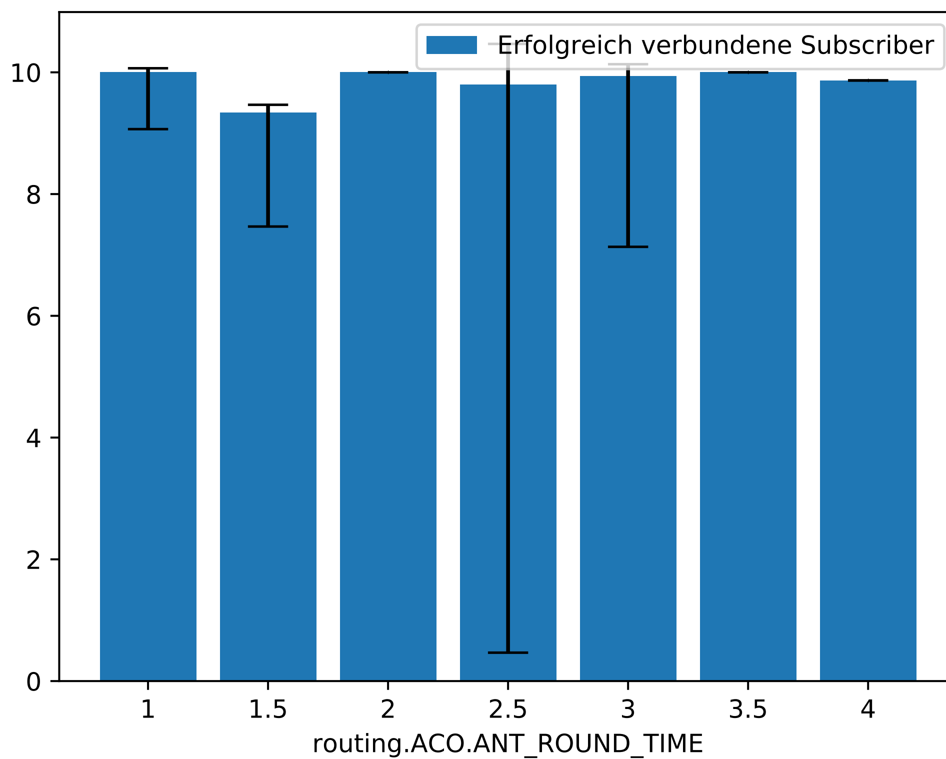


Abbildung 7.7.: Zahl der erfolgreich mit allen Publishern verbundenen Subscriber bei ACO

8 Ausblick

Wie in Kapitel 6 vorgestellt, ist das Framework so flexibel gestaltet, dass das System recht einfach erweitert werden und als Basis für zukünftige Forschung dienen kann. Hierbei kann das System sowohl durch weitere Routingalgorithmen als auch durch neue Anonymisierungstechniken erweitert werden.

Einige mögliche Verbesserungen und Erweiterungen werde ich in diesem Kapitel kurz vorstellen.

8.1 Vollständige Asymmetric Dining-Cryptographers Network Implementierung

Die in dieser Masterarbeit vorgestelle Implementierung legt nur den Grundstein für die Verwendung eines ADC-net um die Anonymität der Publishers zu schützen.

Hierfür sollte das Beispielmodul *routing.CoverTrafficMixin*, welches die Auswahl der ADC-net Knoten über einen Randomwalk durch das P2P Netzwerk zeigt, so erweitert werden, dass diese Funktion über das WebGUI verwendet werden kann. Auch eine Anzeige der ausgebauten ADC-net Verbindungen im WebGUI wäre dann hilfreich.

Ebenfalls eingebaut werden muss die eigentliche Verteilung und anschließende Nutzung der ADC-net Schlüssel im Backend. Hierbei müsste auch eine möglicherweise notwendige Synchronisierung der einzelnen ADC-net Teilnehmer untereinander untersucht werden.

8.2 Routingalgorithmen

Neben den bereits implementierten Routingalgorithmen bietet das Framework auch einfache Möglichkeiten, neue Algorithmen zu implementieren.

Mögliche Kandidaten dafür wären:

Steinerbäume

Obwohl Steinerbäume nur mit globalem Wissen implementiert werden können, bieten sie doch, durch ihre optimale Lösung in Bezug auf die Gesamtlänge aller Pfade, einen guten Vergleichswert für andere Routingalgorithmen.

Eine Implementierung von Steinerbäumen kann also eine gute Grundlage für die Evaluierung anderer Routingalgorithmen bieten.

Neues Hashketten-basiertes Routing

Statt ein Pub/Sub System mit Hashketten-basiertem Routing über das Fluten von Advertisements durch die Publisher aufzubauen, wie das in dieser Arbeit behandelt wurde, könnte man auch die Rollen von Publishern und Subscribern vertauschen, sodass die Subscriber ihr Interesse an einem Topic fluten. Die von den Subscribern gefluteten Advertisements können dann umgekehrt von jedem Publisher jederzeit dafür genutzt werden, eine Datennachricht auf dem aktuell kürzesten Pfad zu allen Subscribern zu schicken, die dem Publisher bekannt sind.

Diese Variante würde ohne den expliziten Aufbau eines Attribute Overlays auskommen, es wird jedoch erwartet, dass die Performance vergleichbar zu dem in dieser Arbeit behandelten Hashketten-basierten Routing von Daubert et al. ist.

Durch die in dieser Variante entfallende Maintenance der aktiven Pfade dürfte dagegen die Zahl der notwendigen Maintenance Messages wesentlich geringer ausfallen. Auch wird die Komplexität des Systems durch diese Variante des Hashketten-basierten Routings vermutlich geringer sein.

8.3 Erweiterung des Webbased Graphical User Interface

Die Benutzeroberfläche des mitgelieferten WebGUI kann an einigen Stellen ebenfalls noch erweitert werden:

Logfenster

Das Fenster mit der Logausgabe könnte um weitere Filtermöglichkeiten nach Thread, Modul oder Datei erweitert werden, die es ermöglichen, gezielt das Verhalten einzelner Komponenten des Backends zu untersuchen. Auch die kombinierte Anzeige der Logmeldungen aller Knoten in einem solchen Fenster könnte nützlich sein.

Scripting Interface

Eine Erweiterung über die man Javascript Code laden kann, der dann Aktionen im WebGUI zeitgesteuert und automatisiert ausführt, würde das Testen neuer Routingalgorithmen und Anonymisierungstechniken erleichtern. Über dieses Scripting Interface ließen sich dann automatisiert Verbindungen zwischen Knoten hinzufügen, Knoten starten oder stoppen, neue Publisher erzeugen etc.

Knotenerzeugung

Im aktuellen WebGUI können zwar Verbindungen manuell erzeugt werden, aber neue Knoten lassen sich nur über eine entsprechende Graph-Datei, aber nicht im WebGUI erzeugen. Die Erweiterung um die Möglichkeit der Knotenerzeugung direkt im WebGUI würde dieses vervollständigen und die komplett manuelle Erzeugung von Graphen im WebGUI erlauben. Diese könnten dann anschließend ebenfalls gespeichert werden.

8.4 Evaluationsframework

Das in dieser Arbeit vorgestellte Framework für Evaluationen kann diese nur lokal auf einem einzelnen Rechner ausführen. Durch einige kleinere Anpassungen ließe sich das Framework aber auch nutzen, um die Knoten des P2P Netzwerks auf unterschiedlichen Rechnern laufen und evaluieren zu lassen.

Hiermit ließe sich auch einfach das Testnetz aus Raspberry Pi Minicomputern direkt evaluieren und dabei gleichzeitig der Datenverkehr mittels der physischen LEDs beobachten.

9 Literatur

[1] Daubert, Fischer, Grube, Schiffner, Kikiras und Mühlhäuser, „AnonPubSub: Anonymous publish-subscribe overlays“, *Computer Communications*, Jg. 76, S. 42–53, 2016, ISSN: 0140-3664. DOI: `https://doi.org/10.1016/j.comcom.2015.11.004` (siehe S. ii, 1 f., 8, 10, 14 f., 17, 25 f., 28, 32 ff., 37).

[2] Grube, Hauke, Daubert und Mühlhäuser, „Ant colonies for efficient and anonymous group communication systems“, in *2017 International Conference on Networked Systems (NetSys)*, März 2017, S. 1–8. DOI: `10.1109/NetSys.2017.7903958` (siehe S. ii, 1 f., 11, 13, 21–24, 32, 42 f.).

[3] Süddeutsche-Zeitung. „Was ist eigentlich bei Facebook los?, Datenmissbrauch“, Süddeutscher Verlag. (5. Apr. 2018), Adresse: `https://web.archive.org/web/20180423001553/https://www.sueddeutsche.de/digital/datenmissbrauch-was-ist-eigentlich-gerade-bei-facebook-los-1.3932349` (besucht am 12. 08. 2018) (siehe S. 1).

[4] Heise-Online. „Rekordhack bei Yahoo war drei Mal so groß“, Heise Medien GmbH & Co. KG. (4. Okt. 2017), Adresse: `https://web.archive.org/web/20180213181944/https://www.heise.de/security/meldung/Rekordhack-bei-Yahoo-war-drei-Mal-so-gross-3849303.html` (besucht am 12. 08. 2018) (siehe S. 1).

[5] Wired.de. „Millionen Sozialversicherungsnummern in den USA gehackt“, Condé Nast Verlag GmbH. (8. Sep. 2017), Adresse: `https://web.archive.org/web/20180221232959/https://www.wired.de/collection/business/hack-equifax-sozialversicherungsnummern-143-millionen` (besucht am 12. 08. 2018) (siehe S. 1).

[6] Tagesschau. „Uber räumt Datendiebstahl ein, Daten von Kunden und Fahrern“, Norddeutscher Rundfunk. (29. Nov. 2017), Adresse: `https://web.archive.org/web/20180730163355/https://www.tagesschau.de/ausland/uber-datenklau-101.html` (besucht am 12. 08. 2018) (siehe S. 1).

[7] D. Chaum, „The dining cryptographers problem: Unconditional sender and recipient untraceability“, *Journal of Cryptology*, Jg. 1, Nr. 1, S. 65–75, 1. Jan. 1988, ISSN: 1432-1378. DOI: `10.1007/BF00206326`. Adresse: `https://doi.org/10.1007/BF00206326` (siehe S. 1 f., 31).

[8] F. Borges, J. Buchmann und M. Mühlhäuser, „Introducing asymmetric DC-Nets“, in *2014 IEEE Conference on Communications and Network Security*, Okt. 2014, S. 508–509. DOI: `10.1109/CNS.2014.6997528`. Adresse: `http://tubiblio.ulb.tu-darmstadt.de/72084/` (siehe S. 1 f., 31).

[9] T. Grube, „Efficient Anonymous Group Communication“, Diss., Technische Universität, Darmstadt, 2018. Adresse: `http://tuprints.ulb.tu-darmstadt.de/7704/` (siehe S. 1 f., 25, 31).

[10] Daubert, Fischer, Schiffner und Mühlhäuser, „Distributed and Anonymous Publish-Subscribe“, in *The 7th International Conference on Network and System Security (NSS 2013)*, J. Lopez, X. Huang und R. Sandhu, Hrsg., Bd. LNCS 7873, Springer, Juni 2013, S. 685–691, ISBN: 978-3-642-38630-5 (siehe S. 2 f., 10, 17 ff., 21, 24, 32 f., 37).

[11] W. Peterson und E. Weldon, *Error-correcting Codes*, Ser. MIT Press. MIT Press, 1972, ISBN: 9780262527316. Adresse: `https://books.google.de/books?id=rFYAjwEACAAJ` (siehe S. 3).

[12] P. T. Eugster, P. A. Felber, R. Guerraoui und A.-M. Kermarrec, „The Many Faces of Publish/Subscribe", *ACM Comput. Surv.*, Jg. 35, Nr. 2, S. 114–131, Juni 2003, ISSN: 0360-0300. DOI: 10.1145/857076.857078. Adresse: http://doi.acm.org/10.1145/857076.857078 (siehe S. 4, 14, 17).

[13] J. Moy, *OSPF Version 2*, RFC 2328 (INTERNET STANDARD), Updated by RFCs 5709, 6549, 6845, 6860, Internet Engineering Task Force, Apr. 1998. Adresse: http://www.ietf.org/rfc/rfc2328.txt (siehe S. 7).

[14] D. Oran, *OSI IS-IS Intra-domain Routing Protocol*, RFC 1142 (Historic), Obsoleted by RFC 7142, Internet Engineering Task Force, Feb. 1990. Adresse: http://www.ietf.org/rfc/rfc1142.txt (siehe S. 7).

[15] G. Malkin, *RIP Version 2*, RFC 2453 (INTERNET STANDARD), Updated by RFC 4822, Internet Engineering Task Force, Nov. 1998. Adresse: http://www.ietf.org/rfc/rfc2453.txt (siehe S. 7).

[16] Schiffner und Clauß, „Using Linkability Information to Attack Mix-Based Anonymity Services", in *Privacy Enhancing Technologies*, I. Goldberg und M. J. Atallah, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, S. 94–107, ISBN: 978-3-642-03168-7 (siehe S. 8).

[17] Pfitzmann und Köhntopp, „Anonymity, Unobservability, and Pseudonymity — A Proposal for Terminology", in *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability Berkeley, CA, USA, July 25–26, 2000 Proceedings*, H. Federrath, Hrsg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, S. 1–9, ISBN: 978-3-540-44702-3. DOI: 10.1007/3-540-44702-4_1 (siehe S. 8).

[18] A. Shikfa, M. Önen und R. Molva, „Privacy and confidentiality in context-based and epidemic forwarding", *„Computer Communications", Elsevier, Vol 33, N°13, August 2010*, Aug. 2010. DOI: http://dx.doi.org/10.1016/j.comcom.2010.04.035. Adresse: http://www.eurecom.fr/publication/3131 (siehe S. 9).

[19] M. K. Reiter und A. D. Rubin, „Crowds: Anonymity for Web Transactions", *ACM Trans. Inf. Syst. Secur.*, Jg. 1, Nr. 1, S. 66–92, Nov. 1998, ISSN: 1094-9224. DOI: 10.1145/290163.290168. Adresse: http://doi.acm.org/10.1145/290163.290168 (siehe S. 9).

[20] Gutjahr, „A Graph-based Ant System and Its Convergence", *Future Gener. Comput. Syst.*, Jg. 16, Nr. 9, S. 873–888, Juni 2000, ISSN: 0167-739X (siehe S. 11, 22, 43).

[21] ——, „ACO algorithms with guaranteed convergence to the optimal solution", *Information Processing Letters*, Jg. 82, Nr. 3, S. 145–153, 2002, ISSN: 0020-0190. DOI: https://doi.org/10.1016/S0020-0190(01)00258-7 (siehe S. 11, 22, 43).

[22] R. Dingledine, N. Mathewson und P. Syverson, „Tor: The Second-Generation Onion Router", in *In Proceedings of the 13 th Usenix Security Symposium*, 2004 (siehe S. 12).

[23] B. Zantout und R. Haraty, „I2P Data Communication System", in *Proceedings of ICN 2011, The Tenth International Conference on Networks*, St. Maarten, The Netherlands Antilles, Jan. 2011 (siehe S. 12).

[24] J. Postel, *Internet Protocol*, RFC 791 (INTERNET STANDARD), Updated by RFCs 1349, 2474, 6864, Internet Engineering Task Force, Sep. 1981. Adresse: http://www.ietf.org/rfc/rfc791.txt (siehe S. 12).

[25] S. Deering und R. Hinden, *Internet Protocol, Version 6 (IPv6) Specification*, RFC 8200 (Proposed Standard), Internet Engineering Task Force, Juli 2017. Adresse: `http://www.ietf.org/rfc/rfc8200.txt` (siehe S. 12, 19).

[26] I. Clarke, O. Sandberg, B. Wiley und T. W. Hong, „Freenet: A Distributed Anonymous Information Storage and Retrieval System", in *International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability*, Berkeley, California, USA: Springer-Verlag, 2001, S. 46–66, ISBN: 3-540-41724-9. Adresse: `http://dl.acm.org/citation.cfm?id=371931.371977` (siehe S. 12).

[27] I. Clarke, O. S, M. Tosel und V. Verendel, *Private Communication Through a Network of Trusted Connections: The Dark Freenet*. Adresse: `https://freenetproject.org/papers/freenet-0.7.5-paper.pdf` (siehe S. 12).

[28] S. Deering, *Host extensions for IP multicasting*, RFC 1112 (INTERNET STANDARD), Updated by RFC 2236, Internet Engineering Task Force, Aug. 1989. Adresse: `http://www.ietf.org/rfc/rfc1112.txt` (siehe S. 13).

[29] J. Postel, *User Datagram Protocol*, RFC 768 (INTERNET STANDARD), Internet Engineering Task Force, Aug. 1980. Adresse: `http://www.ietf.org/rfc/rfc768.txt` (siehe S. 15, 49 f.).

[30] D. Conrads, *Telekommunikation : Grundlagen, Verfahren, Netze*, Wiesbaden, 2004 (siehe S. 34).

[31] A. S. Tanenbaum, *Computernetzwerke*, 4. überarbeitete Auflage. Pearson Studium, Juli 2003, ISBN: 978-3827370464 (siehe S. 38).

[32] Y. Nir und S. Josefsson, *Curve25519 and Curve448 for the Internet Key Exchange Protocol Version 2 (IKEv2) Key Agreement*, RFC 8031 (Proposed Standard), Internet Engineering Task Force, Dez. 2016. DOI: `10.17487/RFC8031`. Adresse: `https://rfc-editor.org/rfc/rfc8031.txt` (siehe S. 50).

[33] J. Postel, *Transmission Control Protocol*, RFC 793 (INTERNET STANDARD), Updated by RFCs 1122, 3168, 6093, 6528, Internet Engineering Task Force, Sep. 1981. Adresse: `http://www.ietf.org/rfc/rfc793.txt` (siehe S. 50).

[34] S. Josefsson, *The Base16, Base32, and Base64 Data Encodings*, RFC 4648 (Proposed Standard), Internet Engineering Task Force, Okt. 2006. Adresse: `http://www.ietf.org/rfc/rfc4648.txt` (siehe S. 51).

[35] A. auf Stackoverflow. „Antwort: Why are Python's 'private' methods not actually private?", Stack Exchange. (16. Sep. 2008), Adresse: `https://stackoverflow.com/a/70900/3528174` (besucht am 16. 09. 2018) (siehe S. 54).

[36] P. Leach, M. Mealling und R. Salz, *A Universally Unique IDentifier (UUID) URN Namespace*, RFC 4122 (Proposed Standard), Internet Engineering Task Force, Juli 2005. Adresse: `http://www.ietf.org/rfc/rfc4122.txt` (siehe S. 60).

[37] Mozilla. „Using server-sent events", Mozilla. (16. Mai 2018), Adresse: `https://web.archive.org/web/20180915121616/https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events` (besucht am 16. 09. 2018) (siehe S. 62).

10 Abkürzungsverzeichnis

P2P Peer to Peer 1

Pub/Sub Publish/Subscribe 1

ACO Ant Colony Optimization 1

TTP Trusted Third Party 14

TTL Time to Live 9

PF Probabilistic Forwarding 26

SG Shell Game 26

RS Rendezvous Subject 31

CRC Cyclic Redundancy Check 3

IoT Internet of Things 13

SPoF Single Point of Failue 14

RTT Round Trip Time 15

DC-net Dining-Cryptographers Network 1

ADC-net Asymmetric Dining-Cryptographers Network 25

RS Rendezvous Subject 31

WebGUI Webbased Graphical User Interface 62

JSON JavaScript Object Notation 49

REST Representational State Transfer 47

SSE Server Sent Events 47

http Hypertext Transfer Protocol 47

AES Advanced Encryption Standard 50

GCM Galois Counter Mode 50

11 Abbildungsverzeichnis

2.1. Ausschnitt einer Hashkette . . . 3
2.2. Pub/Sub Topologien . . . 6
2.3. Vollvermaschtes P2P Netzwerk mit 5 Knoten . . . 7
2.4. Beispiel eines Verteilbaumes . . . 10

4.1. Pfadzusammenfassungen in einem P2P Netzwerk . . . 17
4.2. Relation von Netzwerkdurchmesser und Obergrenze d_{max} . . . 20
4.3. Vergleich von TTL als *Hop Count* und d_{max} . . . 21
4.4. Verschiedene Stadien des Verteilbaumes bei Anwendung von PF . . . 27
4.5. Verschiedene Stadien des Verteilbaumes bei Anwendung von SG . . . 29
4.6. Destruktion des Attribute Overlays . . . 30

5.1. Reflooding von Advertisements . . . 36
5.2. Count to Infinity bei Hashketten-basiertem Routing . . . 39
5.3. Routingtabellen . . . 41
5.4. Versteckter Publisher . . . 44

6.1. Klassendiagramm des Systems . . . 48
6.2. Der Handshake der Connection-Klasse . . . 51
6.3. Die Einstellungen in der Sidebar . . . 65
6.4. Das laufende System mit der Detailansicht eines Knotens in der Sidebar . . . 66

7.1. Dauer des Attribute Overlay Aufbaus bei Hashketten-basiertem Routing . . . 70
7.2. Anzahl der suboptimalen Pfade bei Hashketten-basiertem Routing mit drei Publishern . . . 71
7.3. Anzahl der suboptimalen Pfade bei Hashketten-basiertem Routing mit fünf Publishern . . . 72
7.4. Anzahl der Master und Slaves bei Hashketten-basiertem Routing mit drei Publishern . . . 73
7.5. Anzahl der Master und Slaves bei Hashketten-basiertem Routing mit fünf Publishern . . . 73
7.6. Dauer des Attribute Overlay Aufbaus bei ACO . . . 75
7.7. Zahl der erfolgreich mit allen Publishern verbundenen Subscriber bei ACO . . . 75

12 Tabellenverzeichnis

6.1. Einstellungen der Klasse *Connection* 52
6.2. Statische Methoden der Klasse *Connection* 53
6.3. Instanzmethoden der Klasse *Connection* 53
6.4. Öffentliche Methoden der Klasse *RouterBase* 55
6.5. Private Methoden der Klasse *RouterBase* 56
6.6. Einstellungen der Klasse *Router* 57
6.7. Öffentliche Methoden der Klasse *Router* 57
6.8. Private Methoden der Klasse *Router* 58
6.9. Methoden des Mixins *ActivePathsMixin* 60
6.10. Methoden des Mixins *ProbabilisticForwardingMixin* 61
6.11. Methoden eines Filters 63

13 Listingsverzeichnis

2.1. Abstandsberechnung in Hashketten 4

6.1. Verwendung der Message-Klasse 50
6.2. Aufbau eines Router-Befehls 55
6.3. Beispiel eines Filters 63
6.4. Ausgabe beim Start eines Knotens 64

7.1. Die Einstellung $routing.ACO.ANT_ROUND_TIME$ in JSON-Notation 68

A.1. Vollständiges Beispiel eines Filters 87
A.2. Beispielimplementierung eines Randomwalk-Routers 88
A.3. Kommentierte Struktur eines Taskfiles 89
A.4. Struktur eines Resultfiles 90
A.5. Vollständige Einstellungen der Evaluation 91

Anhang

A Listings

In diesem Kapitel finden sich vollständige Minimalbeispiele für viele in dieser Arbeit erwähnte Konfigurationen oder Codebeispiele.

A.1 Implementierung

Vollständiges Beispiel eines Filters:

```
# *** these filters do NOT see ping messages, nor do routers see them ***
# *** global imports are NOT accessible, do imports in __init__ and bind them to an attribute ***

class Filters(Base):
    def __init__(self, logger):
        # init parent class
        super().__init__(logger)
        # some imports needed later
        self.random = __import__("random")

    def gui_command_incoming(self, command):
        pass

    def gui_command_completed(self, command, error):
        pass

    def gui_event_outgoing(event):
        pass

    def router_command_incoming(self, command, router):
        pass

    def subscribed_datamsg_incoming(self, msg, router):
        pass

    def covert_msg_incoming(self, msg, con):
        pass

    def covert_msg_outgoing(self, msg, con):
        pass

    def msg_incoming(self, msg, con):
        pass

    def msg_outgoing(self, msg, con):
        pass
```

Listing A.1: Vollständiges Beispiel eines Filters

Beispielimplementierung eines Randomwalk-Routers:

```python
import uuid, numpy, logging
logger = logging.getLogger(__name__)

# own classes
from networking import Message
from .router import Router

class Randomwalk(Router):
    settings = {
        "INITIAL_TTL": 60,
        "INITIAL_WALKERS": 5,
    }

    def __init__(self, node_id, queue):
        super(Randomwalk, self).__init__(node_id, queue)
        logger.info("%s router initialized..." % self.__class__.__name__)

    def _route_data(self, msg, incoming_connection=None):
        if msg["ttl"] <= 0:
            logger.warning("Ignoring data because of expired ttl!")
            return

        if self._forward_data(msg):     # inform own subscribers of new data and determine if data is fresh
            logger.info("Data ID already seen, ignoring it and not routing further...")
            return

        msg["ttl"] -= 1
        msg["nodes"].append(self.node_id)
        connections = [key for key in self.connections if key not in msg["nodes"]]     # this does avoid loops
        if not len(connections):
            logger.warning("No additional peers found, cannot route data further!")
            return
        # use at most INITIAL_WALKERS randomly selected peers to send our message to if we are the origin
        # or only 1 peer to pass the message to if we are not the origin
        connections = list(numpy.random.choice(
            connections,
            size=min(len(connections), Randomwalk.settings["INITIAL_WALKERS"] if not incoming_connection else 1)
        ))
        for node_id in connections:
            self._send_msg(msg, self.connections[node_id])

    def _publish_command(self, command):
        # call parent class for common tasks (update self.publishing)
        super(Randomwalk, self)._publish_command(command)
        msg = Message("%s_data" % self.__class__.__name__, {
            "channel": command["channel"],
            "data": command["data"],
            "id": str(uuid.uuid4()),
            "ttl": Randomwalk.settings["INITIAL_TTL"],
            "nodes": []
        })
        self._route_data(msg)
```

Listing A.2: Beispielimplementierung eines Randomwalk-Routers

Das Taskfile:

```
1  {
2      //Einstellungen der Knoten
3      "settings": {
4          //[...] (die Einstellungen sind in einem separaten Listing zu finden)
5      },
6      //Eigenschaften, die für alle Tasks verwendet werden (können in jedem Task auch↩
           einzeln überschrieben werden)
7      "task_defaults": {
8          "base_ip": "127.0.0.100",
9          "nodes": 100,
10         "graph_args": {"avgdegree": 4},
11         "publishers": 3,
12         "subscribers": 10,
13         "rounds": 15
14     },
15     //Liste der eigentlichen Tasks und ihrer Eigenschaften
16     "tasks": {
17         //der Taskname wird bei der Speicherung der Resultate zur Identifikation ↩
               genutzt
18         "flooding_suboptimal_paths": {
19             "runtime": 30,              //Laufzeit der Messung
20             "router": "Flooding",       //Name des verwendeten Routingalgorithmus
21             //Iterator, der die Konfiguration routing.Flooding.SUBSCRIBE_DELAY verä↩
                   ndert (optional)
22             "iterate": {
23                 "setting": "routing.Flooding.SUBSCRIBE_DELAY",
24                 "iterator": "numpy.linspace(0, 4, 9)"
25             },
26             //Initialisierung der Variablen, die im Filter verwendet werden (kann ↩
                   ein leeres Objekt sein)
27             "init": {
28                 "shorter_subscribers": 0,
29                 "shorter_intermediates": 0
30             },
31             //Überführung der Rohvariablen aus dem Filter (deaktiviert den Task, ↩
                   wenn nicht gegeben oder leer)
32             "output": {
33                 "shorter_subscribers": "evaluation.shorter_subscribers",
34                 "shorter_intermediates": "evaluation.shorter_intermediates"
35             },
36             //Zusammenführung der "output" Variablen aus allen Runden (optional)
37             //Wird dieser Block ganz weg gelassen oder eine einzige Variable ↩
                   ausgelassen,
38             //so werden die Rohen Listen der Rundenwerte zurückgegeben
39             "reduce": {
40                 "shorter_subscribers": "numpy.mean(valuelist)",
41                 "shorter_intermediates": "numpy.mean(valuelist)"
42             }
43         },
44         //[...]
45     }
46 }
```

Listing A.3: Kommentierte Struktur eines Taskfiles

Die Ergebnisdatei einer Evaluation:

```
1  {
2      //Liste der ausgeführten Tasks (Tasknamen)
3      "flooding_suboptimal_paths": {
4          //Liste der Iterationen und ihrer Ergebnisse
5          "routing.Flooding.SUBSCRIBE_DELAY = 0.0": {
6              "shorter_intermediates": 0.26666666666666666,
7              "shorter_subscribers": 1.7333333333333334
8          },
9          "routing.Flooding.SUBSCRIBE_DELAY = 0.5": {
10             "shorter_intermediates": 0.3333333333333333,
11             "shorter_subscribers": 1.9333333333333333
12         },
13         //[...]
14         "routing.Flooding.SUBSCRIBE_DELAY = 4.0": {
15             "shorter_intermediates": 0.0,
16             "shorter_subscribers": 0.0
17         }
18 }
```

Listing A.4: Kommentierte Struktur eines Resultfiles

Die bei der Simulation verwendeten Standardeinstellungen:

```
{
    "networking": {
        "Connection": {
        "MAX_COVERT_PAYLOAD": 16384,
        "PING_INTERVAL": 0.1,
        "MAX_MISSING_PINGS": 20,
        "MAX_RECONNECTS": 3,
        "PACKET_LOSS": 0,
        "ENCRYPT_PACKETS": false
        }
    },
    "routing": {
        "Router": {
            "OUTGOING_TIME": 0.02,
            "TIMING_FACTOR": 1
        },
        "ACO": {
            "ANONYMOUS_IDS": false,
            "ANT_COUNT": 10,
            "EVAPORATION_TIME": 4,
            "EVAPORATION_FACTOR": 0.75,
            "DEFAULT_ROUNDS": 100,
            "ACTIVATION_ROUNDS": 5,
            "ACTIVATING_ANTS": 2,
            "ANT_ROUND_TIME": 2,
            "ANT_MAINTENANCE_TIME": 0,
            "AGGRESSIVE_TEARDOWN": false
        },
        "Flooding": {
            "ANONYMOUS_IDS": false,
            "MAX_HASHES": 1000,
            "REFLOOD_DELAY": 3.0,
            "RANDOM_MASTER_PUBLISH": false,
            "SUBSCRIBE_DELAY": 3.0,
            "AGGRESSIVE_TEARDOWN": false,
            "AGGRESSIVE_RESUBSCRIBE": false,
            "MIN_BECOME_MASTER_DELAY": 4,
            "MAX_BECOME_MASTER_DELAY": 10,
            "AGGRESSIVE_REFLOODING": false
        },
        "Randomwalk": {
            "INITIAL_TTL": 60,
            "INITIAL_WALKERS": 5
        },
        "Gossiping": {
            "INITIAL_TTL": 60,
            "GOSSIPING_PEERS": 2
        }
    }
}
```

Listing A.5: Vollständige Einstellungen der Evaluation