FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Coordination-Based Framework for Routing Algorithms in Unstructured Peer-to-Peer Networks

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Stefan Zischka
Matrikelnummer 0828584

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: A.o. Univ.Prof. Dr. Dipl.-Ing. eva Kühn
Co-Betreuung: Dr.techn.Mag. Dipl.Math Vesna Šešum-Čavić

Wien, 19. April 2017

Stefan Zischka

eva Kühn

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Coordination-Based Framework for Routing Algorithms in Unstructured Peer-to-Peer Networks

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Stefan Zischka
Registration Number 0828584

to the Faculty of Informatics
at the Vienna University of Technology

Supervisor: A.o. Univ.Prof. Dr. Dipl.-Ing. eva Kühn
Co-Supervisor: Dr.techn.Mag. Dipl.Math Vesna Šešum-Čavić

Vienna, 19th April, 2017

Stefan Zischka

eva Kühn

Erklärung zur Verfassung der Arbeit

Stefan Zischka
Neustiftgasse 87/1/10 1070 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 19. April 2017

Stefan Zischka

Acknowledgements

First, I would like to thank my supervisors eva Kühn and Vesna Šešum-Čavić for their support throughout the thesis. When writing the thesis, their feedback and guidance was of great value.

Furthermore, I would like to thank Stefan Craß for his feedback in the early development stages of the framework and Stephan Cejka for his support regarding the Peer Model Java implementation.

Most importantly, I would like to show my deepest gratitude to my parents Waltraud and Andreas for their never ending support and their believe in me achieving the goals I have set for myself.

Last but not least, I would like thank all other people, not named explicitly here, who helped me throughout the course of the thesis. I am deeply grateful for their support.

Abstract

The problem of path selection when sending information from one node to another over multiple hops, solved by routing algorithms, is a substantial one in computer networks. Especially in unstructured Peer-to-Peer networks, the topic is of major importance, since no global view on the network or global address mapping exists.

This thesis provides a much needed benchmarking framework that allows the fair and systematic benchmarking and comparison of routing algorithms in unstructured Peer-to-Peer networks. The resulting application, based on the Peer Model (a coordination based programming model), supports easy exchangeability of routing algorithms and extensive configurability.

Additional contributions are the adaption of existing swarm intelligent algorithms from a different domain to the domain of routing. BeeNet is based on the foraging behavior of honey bees, whereas SlimeMoldNet makes use of the Dictyostelium discoideum slime molds life-cycle. Both algorithms are competitively benchmarked, evaluated and compared to five well known routing algorithms: AntNet, BeeHive, Physarum polycephalum routing algorithm, Gnutella Flooding and k-Random Walker. Overall, SlimeMoldNet outperforms the other algorithms in regards to the average data packet delay. This especially holds for bigger P2P network sizes and data packet traffic levels. BeeNet shows similar good results. In terms of scalability, BeeNet outperforms all other algorithms, beside k-Random Walker at some occasions, without having the same major drawbacks. SlimeMoldNets scalability is above average and improves drastically proportional to the network size and data packet traffic level.

Kurzfassung

Die Problemstellung der Pfadselektion beim Senden von Daten über mehrere Netzwerkknoten wird durch Routing Algorithmen gelöst und ist von elementarer Relevanz für die Kommunikation in Peer-to-Peer Computer Netzwerken. Für Peer-to-Peer Netzwerke ist das Routing Problem von spezieller Wichtigkeit, da keine zentrale Sicht auf das Netzwerk und keine globale Adresszuordnung existiert.

Die Diplomarbeit liefert ein dringend benötigtes Framework für faire und systematische Benchmark-Tests von Routing Algorithmen in unstrukturierten Peer-to-Peer Netzwerken und ermöglich deren Vergleich. Die resultierende Applikation basiert auf dem koordinationsbasierten Programmiermodel Peer Model, erlaubt die leichte Austauschbarkeit von Routing Algorithmen und bietet umfangreiche Konfigurierbarkeit.

Ein weiterer Beitrag ist die Adaptierung von zwei vorhandenen schwarm-intelligenten Algorithmen zur Lösung des Routing Problems. BeeNet basiert auf dem Futtersuchverhalten von Honigbienen. SlimeMoldNet hingegen basiert auf dem Lebenzyklus des Dictyostelium discoideum Schleimpilzes. Beide Algorithmen werden kompetitiv gemessen, evaluiert und mit fünf bekannten Routing Algorithmen verglichen: AntNet, BeeHive, Physarum polycephalum Routing Algorithmus, Gnutella Flooding und k-Random Walker. SlimeMoldNet performt ingesamt besser als die anderen Algorithmen bezüglich der durchschnittlichen Datenpaketverzögerung. Dies gilt im Speziellen für größere Netzwerke und große Anzahl von Datenpaketen. BeeNet zeigt ähnlich gute Resultate. Hinsichtlich der Skalierbarkeit performt BeeNet besser als die anderen Algorithmen. Die Ausnahme bildet k-Random Walker in einigen wenigen Fällen, der jedoch dafür in anderen Bereichen wesentliche Mängel aufweist. Die Skalierbarkeit von SlimeMoldNet ist überdurchschnittlich gut und verbessert sich drastisch und proportional zur Netzwerkgröße und Datenverkehr.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

CHAPTER 1

Introduction

In this chapter an introduction to the thesis is provided and the problem statement is discussed. Furthermore, the aim of the master thesis and the methodological approach to achieve these goals are defined. Lastly, the structure of the thesis is provided.

1.1 Problem Statement

The main task of routing algorithms is to solve the problem of path selection when sending information from one node to another over multiple hops within a network (e.g. when sending data packets to a destination node in a computer network) [45].

Thus, in order to send information in a network from a source to a destination, some sort of routing algorithm is needed. It is important to notice that even very primitive approaches, like forwarding a packet to a randomly chosen neighbor, are an instance of a routing algorithm. This makes routing algorithms an essential part of computer networks such as Peer-to-Peer overlay networks.

In fact, for unstructured P2P networks, the topic of routing is of major importance, since no global view on the network exists and no address mapping is maintained. Thus, the delivery of data packets is neither guaranteed nor bound to a specific upper limit of hops [5].

Furthermore, as taxonomy [17] illustrates, routing algorithms exist in a wide range of varieties and can be classified on different characteristics such as the ability to handle changes of the network during run time or the amount of paths taken into consideration when forwarding data packets. This diversity and the resulting substantial amount of different options for their implementation make routing algorithms hard to compare fairly.

Existing benchmarking frameworks focus on specific P2P application protocols like information retrieval [52] or publishing of documents [1]. Therefore, routing is not benchmarked in its abstracted form, but only indirectly, since it is used as a tool by the protocols to fulfill their purpose.
Existing popular network simulators, even when focused on P2P networks (e.g. Peerfact-Sim.KOM [64]), only offer a general environment to simulate distributed systems and do not provide a generic abstracted pattern for benchmarking routing algorithms. Moreover, they often have a steep learning curve and use domain specific programming languages (e.g. NED in OMNet++ [63] or OTcl in NS2 [61]).

Therefore, a generic benchmarking framework for unstructured P2P networks, focused on routing in its abstracted form, is needed. It should enable the systematic and fair comparison of routing algorithms in P2P networks and should provide a meaningful component-based abstraction, able to support any kind of routing algorithm in unstructured P2P networks.

Another class of algorithms in the domain of routing, which have become more and more popular in recent years, are swarm intelligent routing algorithm such as AntNet [13] and BeeHive [49]. In light of this, the possibilities, provided by the generic framework, should be used to adapt two existing swarm-intelligent algorithms, Bee Algorithm [42], used for distributed load balancing, and SMP2P [40], used for search in P2P networks, to the domain of routing.

1.2 Aim of the Work

The aim of the work is to achieve two goals:

- **Benchmarking Framework:** The first major goal of the thesis is to create a benchmarking framework for unstructured P2P networks, that enables the fair and systematic benchmarking and comparison of routing algorithms by providing a meaningful component-based abstraction in form of a pattern. More specifically, the framework should benchmark routing algorithms in a generic manner, stripped from specific areas of applications. Furthermore, the framework shall support the easy exchangeability of routing algorithms and extensive configurability. Additionally, the framework shall be implemented based on the Peer Model [26], a coordination based programming model.
- **BeeNet & SlimeMoldNet:** The second aim of the thesis is to create two routing algorithms by adapting two already existing swarm intelligent algorithms. The first algorithm to adapt is Bee Algorithm [42], an algorithm for solving the distributed load balancing problem, based on the foraging behavior of honey bees. The second algorithm to adapt is SMP2P [40], an algorithm for distributed search in P2P networks, based on the life-cycle of the slime mold Dictyostelium discoideum.
  Furthermore, the framework shall be used to evaluate the created routing algorithm

explained in the following and to compare them to five well known intelligent and non-intelligent routing algorithms.

1.3 Methodological Approach

The following systematic approach will be applied to achieve the defined goals of the thesis:

1. **Literature Review:** This step includes the gathering of information regarding state-of-the-art frameworks, related work and the technical background of the thesis. Furthermore, types of routing algorithms and their diverse characteristics will be analyzed in order to enable the creation of a generic and abstract pattern.

2. **Pattern Modeling:** Using the the gathered information, the modeling of a generic and abstract pattern for routing algorithms in unstructured P2P patterns will be carried out.

3. **Framework Implementation:** Based on the modeled pattern, the benchmarking framework will be implemented using Java 8 [59]. The framework will be based on the Java implementation of the Peer Model [8].

4. **Routing Algorithm Adaption & Implementation:** Next, the swarm intelligent algorithms Bee Algorithm [42] and SMP2P [40] are adapted. The BeeNet and SlimeMoldNet routing algorithms will be implemented in the created framework, alongside AntNet [13], BeeHive [49], a Physarum polycephalum based routing algorithm [19], Gnutella Flooding [28] and the k-Random Walker routing algorithm [30].

5. **Benchmarking & Evaluation:** After determining the best algorithm specific parameters for each of the seven implemented routing algorithms, they will be competitively benchmarked using the created framework. The results will be used to evaluate the performance of the algorithms. Additionally, statistical and scalability analysis will be applied on the result data for further in-depth comparison of the algorithms.

1.4 Structure of the Thesis

The master thesis is structured as follows:
Chapter 2 discusses the related work and states the technical background of the thesis. Chapter 3 defines routing in its abstracted form and contains the descriptions of the contributed routing algorithms BeeNet and SlimeMoldNet. Chapter 4 contains a detailed description of the benchmarking frameworks architecture, its pattern composition and its components. Chapter 5 describes the implementation specifics of the framework, the frameworks configuration and its execution. Chapter 6 defines the benchmark

methodology, evaluates the benchmarks result data and shows the results of additional analysis. Chapter 7 discusses possible future improvements to the proposed framework and the adapted routing algorithms.

CHAPTER 2

Related Work & Technical Background

This chapter describes the technical background of this thesis and its related work. It is structured in four parts.
First, the Peer Model and its graphical notation in this thesis are described.
Then, general information about P2P networks is presented. Next, several aspects of routing in general and in unstructured P2P networks are discussed and specific algorithms are described. Lastly, related state-of-the-art frameworks are discussed and a comparison of the proposed framework to popular network simulators is provided.

2.1 Peer Model

The Peer Model is a coordination based programming model presented by Kühn et al. in [26], [9] and [25]. According to [26], its goal is to satisfy requirements of the design as well as the implementation of (distributed) systems in specific domains.

The top-level component of the Peer Model are peers which exist in Peer Spaces. Peers are designed once and instances of them may be used multiple times. Furthermore, peers may contain nested peers, so called sub peers [26].

Each peer has two containers: a peer-input-container (PIC) in which incoming entries are placed and a peer-output-container (POC) for outgoing entries. Entries are entities which have a specific freely selectable type, may contain data and are used to trigger behavior in the model. The contained data is either used for coordination purposes (co-data) or while conducting behavior (app-data). Behavior is realized by the execution of services [26].

Wirings are contained in peers and execute one or more services. However, the triggering of service execution by wirings is conditional. Thus, only if all conditions are met, a service is executed [26].

Guards represent these conditions in the Peer Model. Additionally, guards may pass entries for processing to the wirings service(s). A guard is created specifically for a specified amount of entries of a specific entry type and has a specific link operation. Furthermore, a guard may makes use of link queries. A link query filters entries based on the contents of their co-data [26].

The available link operations for guards are `read`, `take`, `none`, `test` and `delete`. Both, the `read` and the `take` operation provide one or multiple entries of a specified type to the wirings service(s). However, `take` consumes the entry from the peers PIC, `read` does not. The `delete` operation simply removes an entry from the PIC without providing it to the service. The `none` operation makes sure that no entry of a specific type exists in the PIC. The `test` operation checks if an entry of a specific type is available in the PIC but does not provide it to the service(s) executed by the wiring [26].

Wirings are also able to produce entries when executing services by using actions. When entries are created, a time-to-start (TTS) might be set. The TTS refrains an entry from being processed by a wiring until the TTS has passed [26]. Furthermore, a specific destination peer for a entry can be set via an entries destination property. The entry is then placed by the wiring in the PIC of the set destination peer [26].

In order to enable the modeling and implementation of a more complex control flow, the unique flow identifier (flow-ID) is introduced. Entries may have a flow-ID set at their time of creation. A Wiring may be flow-dependent. Thus, it only processes entries with the same flow-ID, that belong together logically (e.g. for implementing request-response patterns) [26].

It is important to stress, that only the concepts of the Peer Model that are relevant for the thesis were explained in this section.
An example instance of a peer is shown in figure 2.1. The graphical notation used in this thesis to describe the peers and wirings of the framework is based on [10].

A peer, its PIC and POC are represented by gray rectangles. Sub peers have the same notation and are contained in peers. A pink shaded square visualizes a wiring. The service called by the wiring is illustrated by a blue shaded square. The incoming arrows to the left represent the guards of a wiring, the outgoing ones to the right represents its actions. A boolean expression in square brackets represents a guard specific query. Labels on outgoing arrows are TTS values. Bold arrows visualize flow-dependent guards of a wiring.

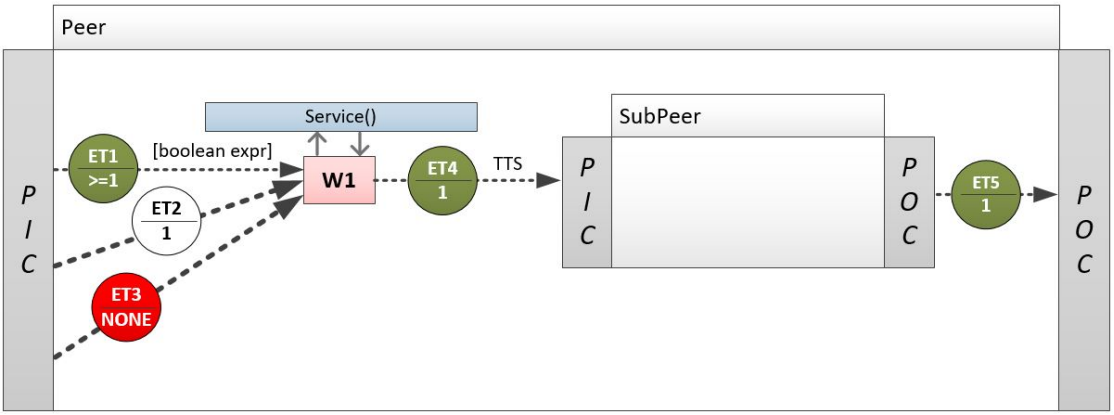

Figure 2.1: Example for a peer, a wiring and a sub peer.

Circles are entries. In the upper half of the circle, the entry type is stated. In the case of a guard, the lower half of the circle states the link count. In the case of an action, it states the amount of entries of the respective type that are produced by the wiring. The shading color of entries represent link operations: green is the take operation, white is the read operation and red is the none operation.
A subpeers incoming arrows and the associated circles represent entries that are placed in the PIC of the subpeer. The ones to the right of a subpeer represents its output (produced by an action of one of the subpeers wirings).

2.2 P2P Systems & Overlay Networks

When approaching the topic of peer-to-peer networks, it is important to grasp the defining properties which makes a distributed system a peer-to-peer one.
While pointing out that there have been a lot of different and debated approaches to the definition of peer-to-peer systems, [2] provides an own definition as follows:

"Peer-to-peer systems are distributed systems consisting of interconnected nodes able to self-organize into network topologies with the purpose of sharing resources such as content, CPU cycles, storage and bandwidth, capable of adapting to failures and accommodating transient populations of nodes while maintaining acceptable connectivity and performance, without requiring the intermediation or support of a global centralized server or authority." [2, p. 337]

This implies the sharing of resources, the avoidance of a centralized server for the general mode of operation, the capability of self-organization and fault resistance the defining characteristics of peer-to-peer systems according to that definition.
Furthermore, [5] states typical characteristics of peer-to-peer systems including the symmetry of roles (each node in the P2P network acts as client and server) and the lack

of global network administration (each node handles its own administration locally). However, [5] also stresses that some of the stated characteristics are not extremely strict i.e. P2P systems may deviate from them, e.g. some P2P systems use nodes with non-symmetric roles.

Due to the easy way of participation, the flexible nature of peer-to-peer systems and the importance of high quality distribution of entertainment media on network devices, peer-to-peer systems have emerged to an important technology for the large scale collection, sharing, publishing and distribution of data [5].

2.2.1 P2P Overlay Networks

P2P overlay networks can be defined as:

"An application layer virtual or logical network in which endpoints are addressable and that provides connectivity, routing, and messaging between endpoints. Overlay networks are frequently used as a substrate for deploying new network services, or for providing a routing topology not available from the underlying physical network. Many peer-to-peer systems are overlay networks that run on top of the Internet." [4, p. 206]

[29] provides an abstract layered architecture of overlay networks that gives a good overview about the components of peer-to-peer overlay networks, their relationship and provided functionalities. As shown in figure 2.2, the architecture is structured in 5 layers. Each of the higher layers is dependent on the ones lower than it in the model. The lowermost layer is the Network Communications layer. It represents the physical network on which the overlay network is built on. One layer above is the Overlay Nodes Management layer, which serves the purposes of routing, location lookup and the discovery of resources in the overlay network. Since this is the layer where routing algorithms operate, the Overlay Nodes Management layer is the most interesting one for this thesis. The middle layer is the Features Management layer. It ensures the robustness of the overlay network by dealing with security and resource management. This also means, that the layer is responsible for the error detection and the mitigation of the impact on the overlays operation. The Service-specific layer supports, as the name suggests, the services of the overlay by managing them (e.g. scheduling of the services), handling the messaging and providing meta-data of resources. The top layer of the abstract model is the Application-level layer. This layer includes all applications, tools and services that run in the P2P overlay network [29].

There are several approaches for the classification of P2P networks based on different properties. In [2], a classification based on the overlay networks distribution of responsibilities and functionalities is provided and describes as follows.
A `fully centralized P2P network` only consists of servlets i.e. all node have equal responsibilities and carry out the same tasks. Thus, no central coordination or data storage unit is used in the networks operation [2].

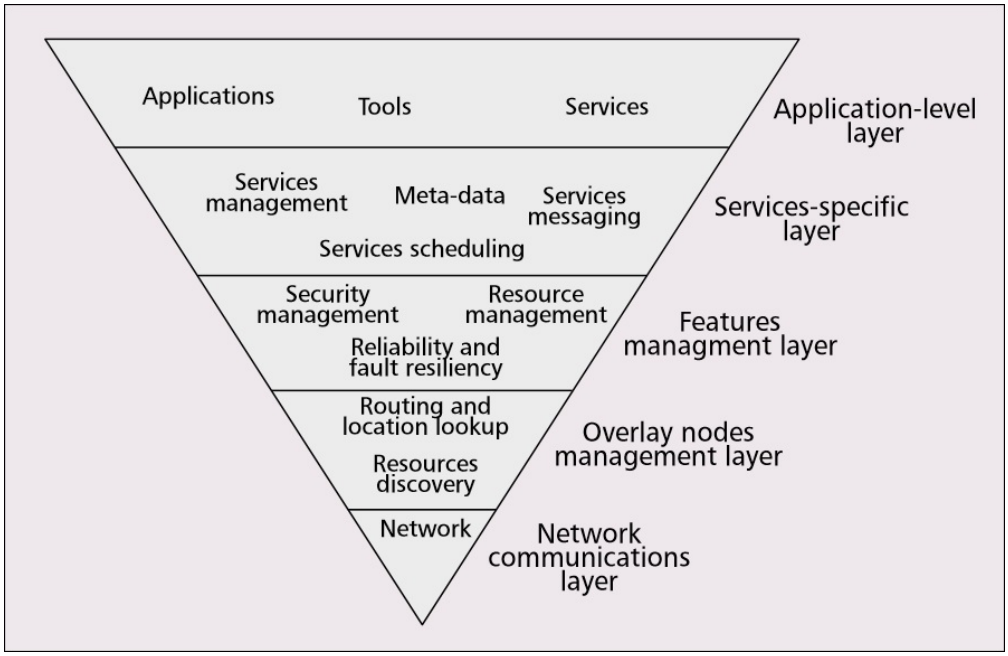

Figure 2.2: An abstract architecture for P2P overlay networks [29]

.

`Partially centralized P2P networks`, do not use a central coordination unit, but assign the role of supernodes to some servlets based on varying criteria. These supernodes then carry out additional functionality to support the coordination and collaboration in the P2P network [2].
`Hybrid centralized P2P networks` use one or more central servers for coordination in the P2P network. Central servers may also store additional (meta)data [2].

Another interesting classification, stated and described by [5], is based on the structure type of a P2P network.
Characteristic for `structured P2P networks` is the usage of routing tables which provide some sort of mapping of resources or unique destination identifiers to specific locations in the P2P network. Moreover, the mapping might be distributed systematically over the P2P network (e.g. usage of distributed hash tables). When a node joins such a P2P network, its routing specific structures are fully initialized and changes to the mapping are propagated accordingly. This mapping allows deterministic routing but also leads to organizational overhead in order to maintain the networks structure [5].
In `unstructured P2P networks`, there is no globally maintained mapping. Network topologies formed by nodes of these P2P networks are often random or scale-free [35], [41]. Therefore, routing is not deterministic but has to be done based on incomplete information. Thus, the delivery of messages is neither guaranteed nor bound to a specific upper limit of hops [5].

2.2.2 Churn

Churn is described in [43] as the dynamic joining and leaving of P2P overlay networks by P2P network nodes and therefore pointed out as a highly significant element of P2P networks dynamic nature. Interestingly, it was shown by analysis of several popular deployed P2P network overlays, that group-level properties of churn (collective behavior of peers) are very similar across different P2P networks, while peer-level properties of churn (individual behavior of peers) differ to a bigger extend. In [38] an in-depth analysis of popular churn models, used for the formal description of churn and to enable its simulation, is provided. More specifically, it considers the models introduced by Yao et al. [51], Leonard et al. [27], Duminuco et al. [15] and Wang et al. [47] as most significant.

2.3 Routing in Unstructured P2P Networks

The main task of routing algorithms is defined in [45] as solving the problem of path selection when sending data over multiple hops from one node to another in a network (e.g. when sending packets to a destination node in a computer network). In this section, design goals and some general classifications of routing algorithms are discussed. Furthermore, a variety of routing algorithms, suitable for routing in P2P networks (some of them already prominently used in P2P protocols), are described.

2.3.1 Design goals

There are some well desired characteristics of routing algorithms, described in [44] as follows. First and foremost, algorithms should work correctly and should be able to handle failures and changes in and of the network. Furthermore they should get and stay in a stable state as quickly as possible, should be as simple as possible and should operate in a fair and efficient manner. Another important desired property of routing algorithms, stated in the Cisco Internetworking Technology Handbook [55], is that routing algorithms should strive to find the most optimal paths for traveling the network. This is done by defining some kind of metric(s) (e.g. number of intermediate nodes visited from starting point to destination) and assigning values of the respective metric to paths of the network. If multiple metrics are used, a routing algorithm may weight them differently for its computation of the most optimal path.

2.3.2 Classification of routing algorithms

Routing algorithms are really diverse i.e. they can differ by a multitude of aspects. This diversity is clearly shown when one examines the taxonomy by Farooq [17], which uses the information stated in the Cisco Internetworking Technology Handbook [55] as basis), that classifies routing algorithms based on their applied policies and strategies such as [17]:

- **Non-adaptive vs. Adaptive:** Distinguishes whether an algorithm is able to take ongoing changes the network (e.g. outage of a node) into consideration for the calculation of routes during run-time.
- **Single-path vs. Multi-path:** Single-path and multi-path routing algorithms differ from each other regarding the number of routes they offer. While the former one only considers one optimal route to a destination, the latter one is not restricted to a single path. Keeping multiple paths for a destination enables load-balancing of traffic over multiple paths.
- **Flat vs. Hierarchical / Intra-Domain vs Inter-Domain:** In Hierarchical routing algorithms nodes are aggregated logically to multiple domains and traffic is routed independently within (Intra-Domain) and between these domains (Inter-Domain) whereas in flat routing algorithms there is only one domain which contains all nodes of the network.
- **Source Routing vs. Hop-by-Hop Routing:** In opposition to source routing, where the entire route from source to destination is already specified at the starting node and the forwarding by intermediate nodes happens according to this specification, routing decisions regarding the next node of the path are made locally on each intermediate node in hop-by-hop routing algorithms.
- **Global vs. Local:** Distinguishes if a routing algorithm has a global view on the network or only partial information on each node in order to make routing decisions.
- **Deterministic vs. Probabilistic:** States whether a routing algorithm always routes traffic to the same destination through the same successive node or if the chosen next hop is based on a probability which is proportional to the goodness of the paths.

Furthermore, routing algorithms can be classified as **non-intelligent** and **intelligent** [41]. The difference is, that intelligent routing algorithms implement some form of reinforced learning by using agents, which interact with the environment (e.g. the network) and receive feedback based on these interactions, in order to make decisions based on the knowledge base (built by the gained experience) [45].
There are several approaches to intelligence in routing algorithms, including algorithms that mimic behavior already existing in the nature (swarm-based intelligence or based on genetic evolution) or machine learning [17].

2.3.3 Routing in Unstructured P2P Networks

It is important to stress that routing algorithms are, unlike the protocols which make use of them, not bound to a specific area of application. Thus, not only routing algorithms prominently used by P2P protocols are relevant, but all routing algorithms that are suitable to route data in unstructured P2P networks. In the following, several routing

algorithms used by protocols in unstructured P2P networks are discussed. Their selection is based on [5]. Moreover, several innovative swarm-intelligent routing algorithms are presented.

2.3.3.1 Gnutella Flooding

As described in [28], the famous Gnutella P2P overlay network makes use of a flooding routing algorithm for forwarding queries. Queries are used to find resources, located at participating nodes in the overlay network.
The decisions made by the algorithm are rather simple. Each data packet, distinguished with an unique identifier, is forwarded to all of a nodes adjacent neighbors, except the one from which the packet was received [28]. However, this strategy results in the problem, that data packets are duplicated and passed through the network, creating huge amounts of traffic. Therefore, Gnutellas flooding algorithm uses an additional measures to mitigate this. It uses a maximum hop count limit for packets, called time-to-live (TTL), which triggers their termination when reached [28].

2.3.3.2 k-Random Walker

The random walk routing algorithm [30] is rather primitive. At any point on their path, data packets are simply forwarded to a random neighbor, until they reach their destination. Thus, the probability for choosing a neighbor $n \in N_c$ at node c as next hop is defined as

$$P(n) = \frac{1}{|N_c|}, \quad N_c = \{neighbors(c)\} \tag{2.1}$$

Since random routing of single data entries is very inefficient, [30] discusses several improvements to the random walker algorithm. Among them is the use of a maximum hop count limit for walkers, called time-to-live (TTL), and the approach to send k instances of a data packet simultaneously instead of a single one. Thus, the improved random walker algorithm is named k-Random Walker [30].

2.3.3.3 AntNet

AntNet, proposed by Di Caro et. al [13], is a swarm intelligent routing algorithm which is inspired by the behavior of ants. A more sophisticated and advanced version of the algorithm is described in-depth in [12] as follows.

The algorithm is based on a phenomenon in nature, observed when ants collect and transport food from a food source to their formicary. At first, ants take random paths to a food source. They communicate indirectly by using a chemical substance, called pheromone, that is left by ants on their path [12]. The shorter the path taken, the faster ants travel back and forth to the food source. Thus, pheromone levels on short paths are higher than those on longer paths. Ants are more likely to choose paths with higher pheromone level, increasing the pheromone level on those paths even more. Therefore,

after some time, the path taken converges to the shortest one [12].

Ants are represented by 2 types of software agents: forward agents, responsible for recording the delay when traveling from the source node to a destination node and backwards agents, responsible for updating data structures of network nodes on their way back. These software agents (in the following called ants) hold a data structure that serves as a history of visited nodes and the time needed to reach them. Furthermore, each network node holds 2 local data structures [12]. The first data structure is a routing table that contains a probability value P_{mn} for each neighbor n of a node and each destination node m in the network. The probability of such a pair (m, n) states how good the decision is to choose n as next hop when m is the destination of the trip (the higher the value, the better the choice). An important constraint is, that the sum of all probability values for a destination m must always be 1 [12]. The trip list is the second data structure contained in every node of the network. It contains the mean μ_m and variance σ_m^2 of trip times which have been experienced by ants when traveling from this node to the destination node m (independently of which neighbor node was chosen as next hop). Moreover, the structure contains the best experienced trip time to destination node m. Those statistical values are used by the backwards ants when updating a nodes routing table [12].

Network nodes periodically choose a destination based on a roulette wheel selection and send a forward ant. The more a destination is requested by received data packets at a network node, the higher is the probability for this destination to be chosen for a newly spawned ant [12]. As a forward ant travels the network independently, it stores information about the experienced delay at each hop in its data structure and chooses the next hop in a probabilistic manner, based on the local routing table and the size of the current nodes output queue [12]. Furthermore, to avoid loops, neighbors which have not been visited are preferably chosen. If a loop occurs nevertheless and thus a forward ant visits a node twice, a cycle in the path of the ant from source to destination is detected. The ant reacts to this by deleting every entry in its internal data structure regarding the involved nodes of the cycle [12]. If a time limit, the time-to-live (TTL), is reached an ant is killed immediately [12]. When a forward ant reaches its destination, its data structure is extracted and passed to a newly spawned backward ant. The forward ant is then killed [12].

The backward ant uses the entries of the forward ants data structure to follow the path taken by the forward ant in reverse. On its way back to the initial source node, the ant updates the trip list and the routing table of each node visited. First, at the current node k, the mean μ_m and variance σ_m^2 for the destination m are updated using a learning value η and the experienced trip time T_m [12]:

$$\mu_m \leftarrow \mu_m + \eta(T_m - \mu_m) \tag{2.2}$$

$$\sigma_m^2 \leftarrow \sigma_m^2 + \eta((T_m - \mu_m)^2 - \sigma_m^2) \tag{2.3}$$

Furthermore, if T_m is better than the best experienced trip time W_m in the observation window w [12]:

$$W_m \leftarrow T_m \tag{2.4}$$

The observation window size w is defined as [12]:

$$w = 5(c/\eta) \tag{2.5}$$

where $c \in [0, 1]$ is a constant.

Then the routing table for the destination m is updated. The probability of the neighbor f, chosen by the forward ant as next hop on the path to destination m, is increased using a reinforcement value $r \in (0, 1]$ [12]:

$$P_{fn} \leftarrow P_{fn} + r(1 - P_{fn}) \tag{2.6}$$

The probabilities of all other neighbors are decreased such that the accumulated probabilities for the destination m is still 1 [12]:

$$P_{hn} \leftarrow P_{hn} - rP_{hn}, \quad h \in N_k, \quad N_k = \{neighbors(k)\}, \quad h \neq f \tag{2.7}$$

The reinforcement value r is calculated by [12]:

$$r = c_1\left(\frac{W_m}{T_m}\right) + c_2\left(\frac{I_{sup} - W_m}{(I_{sup} - W_m) + (T_m - W_m)}\right) \tag{2.8}$$

r is saturated at 0.9. c_1 and c_2 are constant values [12]:

$$I_{sup} = \mu_m + z\left(\frac{\sigma_m}{\sqrt{w}}\right) \quad with \quad z = 1/\sqrt{(1 - \gamma} \tag{2.9}$$

$\gamma \in [0.6, 0.8]$ is a constant value. Additionally, before the reinforcement value is used, it is squashed using the formula [12]:

$$r \leftarrow \frac{s(r)}{s(1)} \tag{2.10}$$

$$s(x) = \left(1 + exp\left(\frac{a}{x|N_k|}\right)\right), \quad x \in (0, 1] \tag{2.11}$$

a is a constant value.
For forwarding, AntNet offers multiple routes. The route is chosen in a probabilistic manner based on the probability values for a destination in a nodes routing table [12].

2.3.3.4 Physarum polycephalum Algorithm

Several routing algorithms [19], [21], [53] based on the Physarum polycephalum slime mold have been proposed. The algorithm, proposed by Hickey et. al [19], is interesting for routing in unstructured P2P networks, since it allows routing on incomplete information.

The Physarum polycephalum slime mold contains a venous structure that is used for transporting nutrients from food found by the organism. If food is found by an area of the slime mold, the flow in the veins that transport nutrients increases, resulting in increased diameter of those veins. Meanwhile, the diameter of non-nutrient transporting veins decreases. Initially, all diameters of the slime molds veins, and therefore all flows, are considered to be equal [19].

Each network node k holds a routing table containing the flow Q_{ij} in the connecting vein for each neighbor $i \in N_k$ and destination j in the network. The flow represents the probability to choose the vein to a neighbor when sending a data packet to a destination. An important constraint is, that the sum of all flow (probability) values for a destination m must always be 1 [19].

Nutrients are embodied by agents which are periodically spawned and sent to random destinations in the network. Agents have a forward and a backward mode. In forward mode, the agents try to reach their destination by choosing the next hop probabilistically, based on the flow values in the routing table, at each intermediate node. When the destination is reached, an agent goes into backward mode. In backward mode, the agents follow the path, taken by the agent in forward mode, in reverse and update the routing table of each intermediate node using the delta function [19]:

$$\Delta f\Big(Q_{ij}\Big) \leftarrow \epsilon\Big(Q_{ij}\Big) \tag{2.12}$$

The algorithm ignores the delay experienced by forward agents and uses a constant reinforcement value r for updating routing tables [19]:

$$r = \epsilon, \quad \epsilon \in (0, 0.1] \tag{2.13}$$

Forwarding of data packets is done probabilistically by a roulette wheel selection based on the vein diameter ratio [19].

2.3.3.5 BeeHive

Another nature inspired routing algorithm which uses swarm intelligence, is BeeHive, proposed by Wedde, Faroo and Zhang in [49]. It is exhaustively described in [18] as follows. The algorithm is based on the communication techniques bees use when evaluating food sources. A waggle dance is performed in the hive by forager bees in order to communicate food sources of good quality. Hence, bees use this form of communication to maximize their productivity when collecting food. In the BeeHive algorithm, routing tables, held

by each network node, are considered the dance floor in the hive [18].

There are 2 types of bee agents, short distance bees and long distance bees, which differ by the amount of hops they are allowed to travel in the network before they are killed. The algorithm goes through an initialization phase before entering normal mode of operation. In this phase, the network is partitioned into foraging regions. Therefore, nodes send short distance bees to their neighbors in order to elect a representative node. The size of foraging regions is limited by the number of hops a short distance bee is able to travel. Each network node has an unique identifier and the one with the smallest ID wins the election. Representative nodes differ from other nodes by the type of bees they send. They only send long distance bees, whereas all other nodes only send short distance bees. At the end of the initialization phase, all nodes in the network inform each other to which foraging region they belong using long distance bees [18].

Each node in the network contains 3 routing tables, where the mimicked waggle dance is performed in order to exchange path information. The first one is a mapping structure, called Foraging Region Membership routing table (FRM). It contains a mapping of every known node to its foraging region. The Intra Foraging Zone routing table (IFZ) and the Inter Foraging Region routing table (IFR) have the same structure, but are used differently. They contain for each neighbor node and destination a pair (p, q), where p is the propagation delay and q the queue delay. This pair describes the experienced delays when choosing the neighbor as next hop for this particular destination. The IFZ is used for routing data to destinations within the foraging zones, while the IFR holds the same information for the networks representative nodes [18].

Each node periodically spawns a bee agent (long or short distance depending on the role of the node), with an unique identifier, that is then flooded over the network. When a bee reaches a node on its path, it is first checked if this exact copy of the bee has already been received and if the maximum hop count of the bee has been reached [18]. If either one is the case, the bee is killed. Otherwise, it updates either the IFZ or IFR routing table, depending on its type, using the experienced queue delay q_{bee} and propagation p_{bee} for destination r and the last hop neighbor s [18]:

$$q_{sr} = q_{bee} \tag{2.14}$$

$$p_{sr} = p_{bee} \tag{2.15}$$

If a flooded duplicate of a bee has already been received, it is killed after the routing table update. As bees travel the network, at the current node k, they store the accumulated experienced queue delay q_{bee} and propagation delay p_{bee}, weighted by a goodness factor g, for the current hop and all neighbors $n \in N_k$ in their internal data structure [18]:

$$q_{bee} = \sum_{n \in N_k} q_{nr} g_{nr} \tag{2.16}$$

$$p_{bee} = \sum_{n \in N_k} p_{nr} g_{nr} \tag{2.17}$$

Furthermore, if no copy of this bee has been received by the node, it creates replicates of it and sends them to all neighbors except the one from which the bee came from [18].

The goodness g_{jd} of reaching a destination node d over the neighbor node j from the current node is approximated by the formula [18]:

$$g_{jd} = \frac{\frac{1}{p_{jd}+q_{jd}}}{\sum_{k=1}^{N}(\frac{1}{p_{kd}+q_{kd}})} \tag{2.18}$$

where p_{jd} and q_{jd} are the propagation delay and queue delay when d is the destination and j is chosen as next hop node. N is the total number of neighbors. For the forwarding of data packets at a node, BeeHive calculates the goodness function for every possible next hop to the destination of the packet and forwards the received packet in a probabilistic manner, proportional to the calculated goodness values. The delay values are taken from the IFZ, if the destination is in the foraging region of the node. Otherwise, the representative node of the destination is determined using the FRM. The delay values are then read from the IFR [18].

2.4 Related Work

A benchmarking framework is a general abstraction of the target domain, provides generic functionality and allows the testing and finetuning of algorithms in order to enable the selection of the most optimal algorithm for specific scenarios [42].
The main requirements and desired characteristic for the framework are to enable the fair and systematic benchmarking and comparison of routing algorithms by providing a meaningful component-based abstraction in form of a pattern in a generic manner (stripped from a specific area of application like information retrieval and able to support any kind of routing algorithm). Furthermore, the framework should focus on unstructured peer-to-peer networks and should support the easy exchangeability of routing algorithms.

In the following, relevant related state-of-the-art frameworks for routing in unstructured P2P networks are discussed and evaluated with respect to the characteristics discussed above.

Agosti et al. [1] propose a framework, built on top of a P2P network simulator, for simplifying the development of routing protocols and the benchmarking of their performance. It supports unstructured P2P networks. However, routing algorithms are not benchmarked in an isolated and generic manner but in the specific application of information retrieval. Therefore, the framework is not considered generic by the principles stated at the beginning of this section. For the implementation of routing protocols, only an interface with a method for the publishing and for the search of resource descriptors is provided, which makes benchmarked protocols easily exchangeable. Moreover, the framework provides building blocks (such as a topology constructor and a initializer of

nodes). However, no specific pattern for the implementation of routing algorithms is provided and therefore the benchmarking is not considered fair in terms of the principles stated above.

Zammali and Khedija [52] provide a framework for the benchmarking of routing algorithms in combination with distribution and replication strategies in the specific context of information retrieval/distribution of documents in (unstructured) peer-to-peer networks. Although the framework is structured in a modular way (e.g. modules for replication and distribution of documents), the routing algorithms are not structured in a generic pattern. Therefore, the benchmarking is not considered fair based on the stated criteria. It is assumed, that no easy exchangeability of routing algorithms is provided, since it neither claimed, nor any details on the implementation of routing algorithms is given in the paper.

Cuzzocrea [11] presents a standalone configurable query-strategy-based benchmarking framework for information retrieval protocols. The focus is to benchmark the accuracy and efficiency of queries. Again, routing algorithms are only benchmarked indirectly in the context of queries for information retrieval in (unstructured) peer-to-peer networks. The framework strives to be generic in respect of information retrieval techniques by offering complete customizability and the possibility to implement any query algorithm. However it is not generic regarding routing algorithms by the stated criteria, since routing algorithms are not benchmarked in an isolated and abstract manner. Although an interface for the implementation of protocols is provided to ease the exchange of protocols, no specific pattern for the implementation if provided. Therefore, the framework is not considered to offer a fair comparison.

Although, the framework provided by Saleem, Di Caro and Farooq in [37] is not focused on benchmarking of routing algorithm in unstructured P2P networks and therefore only partly relevant for this section, it is still worth mentioning, since it provides an important desired characteristic for the desired framework, a meaningful abstraction of swarm intelligent routing algorithms in form of a component-based break down.

In summary it can be said, that after extensive literature research, no frameworks for the benchmarking of routing algorithms in unstructured peer-to-peer networks could be found that satisfy all the defined main requirements of genericity, fairness and easy exchangeability of routing algorithms.

Characteristics	PeerSpace Framework	[1]	[52]	[11]	[37]
Benchmarking framework	✓	✓	✓	✓	✗
Generic	✓	✗	✗	✗	✗
Pattern for routing algorithms provided	✓	✗	✗	✗	✓
Peer-to-Peer	✓	✓	✓	✓	✗
Fair comparison of algorithms	✓	✗	✗	✗	✗
Easy exchangeability of algorithms	✓	✓	✗	✓	✗
Standalone	✓	✗	✗	✓	✓

Table 2.1: Evaluation of framework characteristics. Each characteristic is marked with ✓, if it is met by the framework. If that is not the case, it is marked with ✗.

2.4.1 Network Simulators

[14] defines a simulator as a piece of software that mimics the real world through the creation of models of the system of interest, which are then used to investigate the behavior of the system such that conclusions can be drawn regarding the real world.
Network simulators narrow down their focus to the simulation of computer and communication networks by modeling their devices (such as routers and hubs), their behavior and the interaction between them [32]. A popular scenario of application for network simulators is the analysis and benchmarking of new algorithms and protocols as well as their comparison to existing counterparts [39].
There are several kinds of network simulators, some of them with different architecture and area of application.
The list of popular simulators contains software such as OMNet++ [63] and J-SIM [58], which are general purpose simulators (not bound to simulation of telecommunication networks), NS-2 [61] and NS-3 [62], which are network simulators (not bound to P2P networks) and specialized P2P network simulators like PeerfactSim.KOM [64].

2.4.1.1 Network simulators vs. the Peer Space framework

There are some downsides of using network simulators when it comes to benchmarking routing algorithms.

- Generally, network simulators offer an environment for the simulation of systems which is not specifically focused on benchmarking routing algorithms and is not restricted enough in terms of how routing algorithms have to be implemented

besides the basic underlying architecture and interfaces of the simulator. They offer no concrete pattern like the proposed framework in this thesis does).

- A further major downside when using a network simulator is their steep learning curve. Not only does one have to read an immense amount of documentation and tutorial pages in order to use them correctly, when it comes to the implementation of own simulation models (e.g. routing algorithms), even more learning time is needed. Moreover, one might even has to learn a new programming language to create simulations in a network simulator. (e.g. NED in OMNet++ [63] or OTcl in NS2 [61]).

- Compared to the framework proposed in this thesis, network simulators often benchmark the algorithms locally, whereas the Peer Space framework can also be used as a real testbed due to the distributed nature of Peer Spaces.

- Another benefit of using the Peer Space framework over a network simulator is that due to the provided pattern, one is able to exchange certain parts of the algorithm easily, which not only gives better insight on how certain parts of the algorithm influence the overall performance, it also serves as a tool for guidance when it comes to the development of new or improved routing algorithms.

2.5 Summary

In this chapter, the technical background and the related work of this thesis is presented. First, the Peer Model, a coordination-based programming model, which combines the domains of design and implementation, is discussed. This fact and the models structured nature makes it perfectly suitable for the implementation of a generic benchmarking framework for routing algorithms.
The rest of the technical background focuses on P2P networks and on routing algorithms. In addition, several specific routing algorithms, suitable for routing in unstructured P2P networks are presented. The diversity of routing algorithms and the flexible nature of unstructured P2P networks emphasizes the need for a abstracted generic benchmarking framework provided in this thesis.

The related work is focused specifically on related state-of-the-art frameworks for routing in unstructured P2P networks. It is shown, that none of the related framework fulfills the desired characteristics of an abstracted generic benchmarking framework for routing in unstructured P2P networks. Additionally, a short comparison of this thesis framework to popular network simulators is provided. Several sophisticated arguments are presented for using the Peer Model framework, provided in this thesis, instead of popular state-of-the-art network simulators.

CHAPTER 3

Bio-inspired Algorithms for Routing in Unstructured P2P Networks

This chapter contains the general definition of how routing in unstructured P2P networks, in its abstracted form, is interpreted in this thesis. Furthermore, adaptions of two well known swarm intelligent optimization algorithms for the purpose of routing in unstructured P2P networks, are presented. Those adaptions are contributions to this master thesis.
The algorithms are both described in three parts. First, their source in nature is discussed. Then, the process of creating their knowledge base (routing table) by applying swarm optimization is described. Lastly, the forwarding of data packets is specified.

3.1 Routing in Unstructured P2P Network Definition

Since routing in unstructured P2P networks is often used in protocols for specific applications, an abstracted model for routing in unstructured P2P networks has to be defined. Generally, the goal is to deliver a data packet to its destination node, defined by its source node. The process is considered successful, if the data entry reaches its destination.

To enable a source node to address the destination of a data packet explicitly, an unique identifier of each P2P node of the network has to be known. However, since P2P overlay networks operate above the physical layer, this unique identifier must not be the network nodes physical host address. Otherwise, the use of routing in the unstructured P2P overlay would be absurd, since nodes would communicate directly.

Let an unstructured P2P network be represented by an undirected graph $G_{P2P} = (V_{P2P}, E_{P2P})$, where the vertices $v \in V_{P2P}$ of the graph represent the P2P networks nodes and the edges $e \in E_{P2P}$ represent connections between these nodes. Nodes $w, k \in V_{P2P}$ are considered neighbors if and only if $\exists(w, k) \in E_{P2P}$.
Each node $v_i \in V_{P2P}, i = [1, n], n = |V_{P2P}|$ has a logical unique identifier x_i and a physical address y_i. x_i of node v_i is known to all nodes $v \in V_{P2P}$. However, only neighbors are able to to map their logical identifier x to their physical host address y and therefore exchange packets directly.
Thus, the address resolution function for a network node c and a unique identifier x_e

$$m(c, x_e) = \begin{cases} y_e & \text{if } e \in \text{neighbors(c)} \\ x_e & \text{otherwise} \end{cases} \tag{3.1}$$

evaluates if node c is able to determine the physical address of node e and therefore if it is able to communicate directly with it. However, it is important to note, that if intelligent agents are used by a routing algorithm, these agents may know the physical address y_s of their source node in addition to the logical identifier x_s. Thus, they may return directly to their source.

Example 3.1.1 *Suppose an instance of an unstructured P2P network represented by the undirected graph* $G_{exP2P} = (V_{exP2P}, E_{exP2P})$ *with* $V_{exP2P} = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$ *and* $E_{exP2P} = \{(v_1, v_2), (v_1, v_3), (v_1, v_5), (v_1, v_6), (v_1, v_7), (v_2, v_3), (v_2, v_4), (v_3, v_4), (v_3, v_5), (v_3, v_6), (v_6, v_7)\}$. *It is shown in figure 3.1.*
The logical unique identifiers known to node v_7 *are* $x_1, x_2, x_3, x_4, x_5, x_6, x_7$. *The physical addresses known to node* v_7 *are* y_1, y_6, y_7. *Thus, direct exchanging of data packets is only possible with nodes* v_1 *and* v_6. *For all other nodes, packets have to be delivered by using a routing algorithm.*
However, if an intelligent agent, originated by another node than v_1 *and* v_6 *reaches* v_7, *it may also return directly to its source.*

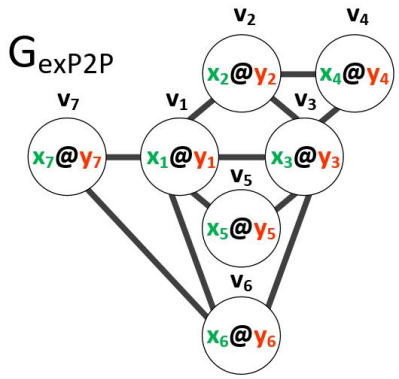

Figure 3.1: Example instance of an unstructured P2P network, visualized as undirected graph

3.2 BeeNet

Šešum-Cavic and Kühn [42] present Bee Algorithm, a swarm intelligent algorithm, created for solving the problem of dynamic load balancing in distributed systems, based on the biological behavior of bees in nature. Moreover, this load balancing algorithm, adopts some elements of Bee Colony Optimization (BCO) described in [50]. It is used for solving the Traveling Salesman Problem. Bee Algorithm, and therefore elements of BCO, are adapted to the domain of routing in unstructured P2P networks. The resulting routing algorithm is called BeeNet.

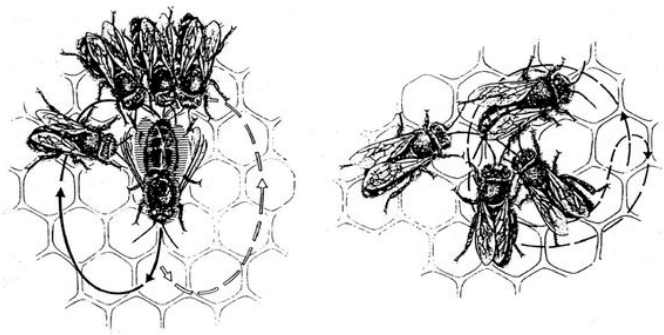

Figure 3.2: Communication of honey bees via dancing [46]

.

Honey bees have an interesting collaborative approach to feed the hive. The central phenomenon of this collaboration is the communication of food source related information via a dance-language known as waggle-dance [46].
Honey bees, that are responsible for searching and collecting food, can switch between 2 kinds of roles: forager and follower. Foragers scout the landscape surrounding the hive and search for good quality sources of nectar and pollen [6].

After finding such a food source and returning to the hive, they unload the collected food and start to waggle-dance. Thus, they communicate the quality level, the distance and the direction of the food source to other bees. Bees observing the dance (foragers and followers) decide which role they take next. They either become followers and follow the dancing forager to the advertised food source or become foragers and scout for food sources themselves [6].

3.2.1 BeeNet for Routing in Unstructured P2P networks

The unstructured P2P network represents the landscape in which many hives and flowers exist. More specifically, each node in the P2P network represents exactly one hive with its own bee population and exactly one flower that contains nectar. Each hive periodically, in an configurable `beeSpawnInterval`, sends out a bee for the search of food to a randomly but specifically chosen flower. It is important to note, that although

only a single bee is sent out by a hive at a point in time, bees explore the network concurrently. Thus, the amount of exploring bees of a hive can be freely configured by the `beeSpawnInterval`. The goal of a hives bee population is to maximizes its productivity and therefore to find the most efficient paths from the hive to all flowers in the landscape.
The quality of a specific path is defined by a fitness function

$$f_{path} = \left(\frac{1}{H_{path}}\right)\left(\frac{1}{D_{path}}\right) \tag{3.2}$$

where H_{path} is the amount of hops a bee took on the path from source to destination and D_{path} is the accumulated delay experienced on this path.

The behavior discussed above, can be described, like the bee based algorithms presented in [42] and [50], as Bee Colony Optimization meta heuristic, shown in pseudocode 3.1. It is described in detail in the following subsections.

Algorithm 3.1: Bee Colony Optimization model for routing

```
Input  : hive s, flowers R
Output : optimal paths from s to all flowers r ∈ R
1 procedure BCO (s, R);
2 Initialization;
3 while not terminated do
4     if beeSpawnInterval has passed then
5         ObserveWaggleDance;
6         ConstructSolution;
7         PerformWaggleDance;
8     end
9 end
```

3.2.2 Initialization

Before a hive sends out bees to search for food, it is initialized. Routing decisions in a hive are made based on two data structures. The routing table RT, held locally at a hive, represents the dance floor on which bees advertise paths to specific flowers. Therefore, it is a collection of paths, experienced by the hives bees, from the hive to destination nodes in the network. The class diagram of an advertised path is shown in figure 3.3. Furthermore, a hive holds a distance matrix M that contains the heuristics distances to neighbor hives. The heuristics distance to a neighbor hive is the delay last experienced by a bee.

A bee is represented by a software agent, shown in figure 3.4. It has exactly one source

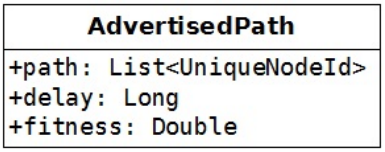

Figure 3.3: Path advertised by a waggle-dancing bee

hive and one destination flower. Moreover, it is important to note that the source address of a bee is the host address of its hive and not its unique identifier in the P2P network. Thus, the bee is able to return to its hive directly. Furthermore, the bee holds a list of trip records which represents the path taken by the bee. An element of this list, shown in figure 3.5, contains an unique identifier of a visited node and the experienced delay. Additionally, if the bee is a follower, it uses a preferred path, which is followed.

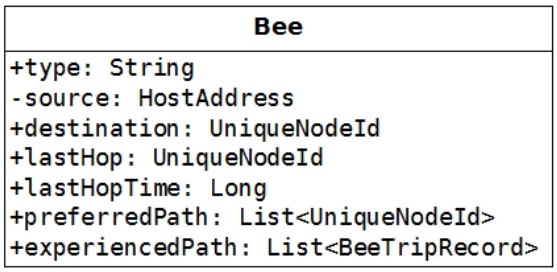

Figure 3.4: Class diagram of a BeeNet bee

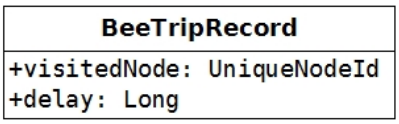

Figure 3.5: Trip memory of a BeeNet bee

Furthermore, before sending out the first forager bees, a hive first sends test bees to its neighbors. They immediately return after reaching the neighbor and set the heuristic distance in the source hives heuristic distance matrix M.

3.2.3 Observe Waggle Dance

Before a bee leaves the hive, its destination flower and its role is determined. At the beginning, no waggle-dances are performed on the dance floor (the routing table does not contain any paths). Thus, all bees automatically take the role of a forager. The target flower r is chosen randomly. However, if for the chosen destination, paths are advertised by waggle-dancing, a bee goes through a specific decision process. First, the bee selects a

advertised path with probability [42]:

$$P_{sel}(path) = \frac{f_{path}}{f_{colony}(r)} \tag{3.3}$$

$$f_{colony}(r) = \frac{1}{n} \sum_{i=1}^{n} f_{r,i} \tag{3.4}$$

where f_{colony}(r) equals the average fitness function value of all currently advertised paths for a specific destination flower r and n is the number of advertised paths to r [42].

After a path has been chosen, the bee evaluates it against $f_{colony}(r)$. The probability to become a follower bee and therefore follow the chosen preferred path is defined in table 3.1.

Evaluation Result	P_{follow}
$f_p < 0.5 f_{colony}(r)$	0.60
$0.5 f_{colony}(r) \leq f_p < 0.65 f_{colony}(r)$	0.20
$0.65 f_{colony}(r) \leq f_p < 0.85 f_{colony}(r)$	0.02
$0.85 f_{colony}(r) \leq f_p$	0.00

Table 3.1: BeeNets lookup table for the probability to become a follower bee [34]

.

The procedure is described in pseudocode 3.2.
First, the source of the bee is set and destination is determined by using the method `selectRandomDestination`. Furthermore, the `lastHopTime` of the bee is initialized to the current time (method `Time.now`). If no path exists for the selected destination in the routing table, a bee automatically becomes a forager. Otherwise, a path is selected by the `selectedPath` method as defined in equation 3.3. After calculating the `colonyFitness` by using the `calculateColonyFitness` method, the `evaluatePath` method is called, to check if the bee will follow the `selectedPath`. If that is the case, the bee becomes a follower and its `preferredPath` is set to the `selectedPath`. Otherwise, the bees `type` is set to forager.

3.2.4 Construct Solution

A bee hops from node to node on its travel from the hive to the destination flower. At each visited node, a bee stores the nodes unique identifier x and the experienced delay in its memory. Thus, back at the bees hive, the path and experienced delays can be fully reconstructed later on. If a bee experiences a loop, and therefore visits a node on the path twice, it erases all information of the loop from its memory.

Algorithm 3.2: BeeNet ObserveWaggleDance procedure

```
   Input  : source hive h, possible destinationNodes
   Output : initialized bee
 1 procedure ObserveWaggleDance (h, destinationNodes);
 2 bee.source ← h;
 3 randomDestination = selectRandomDestination(destinationNodes);
 4 bee.destination ← randomDestination;
 5 bee.lastHopTime ← Time.now();
 6 if no waggle-dance performed for bee.destination then
 7 |   bee.type ← forager;
 8 else
 9 |   selectedPath = selectPath(r);
10 |   colonyFitness = calculateColonyFitness(r);
11 |   becomeFollower = evaluatePath(selectedPath , colonyFitness);
12 |   if becomeFollower == true then
13 |   |   bee.type ← follower;
14 |   |   bee.preferredPath ← selectedPath;
15 |   else
16 |   |   bee.type ← forager;
17 |   end
18 end
```

The transition rule $P_{ij}(t)$ is used to determine the next hop j of a bee at node i and time t [42], [50]:

$$P_{ij}(t) = \frac{[\rho_{ij}]^{\alpha}[1/d_{ij}]^{\beta}}{\sum_{j \in A_i(t)} [\rho_{ij}]^{\alpha}[1/d_{ij}]^{\beta}} \tag{3.5}$$

where d_{ij} is the heuristic distance between the current node i and node j, contained in the heuristic distance matrix M, and ρ_{ij} is the arc fitness of i and j. The constants α and β weight these values in the transition rule. $A_i(t)$ is the set of neighboring nodes of the current node i without the last hop node of a bee. The last hop of a bee is only considered as next hop if it is the only neighbor of i.

The arc fitness p_{ij} is calculated differently for forager and follower bees.
For forager bees the arc fitness function [42], [50]:

$$\rho_{ij} = \frac{1}{k} \quad k = |A_{i(t)}| \tag{3.6}$$

is used.
However, for follower bees, a different arc fitness function 3.7 is applied [42], [50].
The probability to choose a neighbor node j at the current node i as next hop is λ, if j is the next hop on the followers preferred path $F_i(t)$. However, with probability $(1 - \lambda)$, a

follower can break out of following the preferred path and to choose one of the neighbors $j \in A_i(t) : j \notin F_i$. If the follower bee leaves the preferred path, the chance of selecting one of the other neighbors is equally distributed. The follower bees then transitions with arc fitness function 3.6, until it reaches its destination or until it crosses the preferred path again [42], [50]:

$$\rho_{ij} = \begin{cases} \lambda & \text{if } j \in F_i(t) \\ \dfrac{1 - \lambda\ |A_i(t) \cap F_i(t)|}{|A_i(t)| - |A_i(t) \cap F_i(t)|} & \text{otherwise} \end{cases} \qquad \forall j \in A_i(t), 0 \leqslant \lambda \leqslant 1 \tag{3.7}$$

If a bee has reached its destination, it returns to its hive directly in a P2P way.

The pseudocode of the `ConstructSolution` procedure is described in 3.3.
When a bee reaches its destination, it returns to its source node by performing the `ReturnToSource` method and executing the `PerformWaggleDance` procedure.
Otherwise, if the bee is of type forager or no neighbor of the current node is part of the followed `preferredPath`, the bees `nextHop` is selected using the method `applyForagerTransitionRule`. For all other follower bees, the `nextHop` is selected by calling the `applyFollowerTransitionRule` method.
If the current node has already been visited, a loop is detected and deleted by the `deleteLoop` method. When no loop is detected, the current node and the time passed since the last hop of the bee is added to the bees memory `experiencedPath`. After that, the bee transitions to the next hop node through calling the `MoveToNode` method and the `ConstructSolution` procedure is executed again.

3.2.5 Perform Waggle Dance

When a bee returns to its hive, it evaluates the experienced path and sub-paths against the table 3.1 using fitness function equations 3.2 and 3.4. If the evaluation of a path succeeds, it is added to the nodes routing table.

The procedure is described in pseudocode 3.4.
First a reference to the nodes routing table is read by calling the `getRT` method. After the nodes `colonyFitness` is calculated by the `calculateColonyFitness` method, the `experiencedPath` to the destination and all contained sub-paths to the intermediate nodes of this `experiencedPath` are evaluated, as described above, by calling the `evaluatePath` method. The behavior is illustrated in example 3.2.1. If an evaluated paths fitness value is good enough, according to table 3.1, it is added to the nodes routing table `RT`. Otherwise, the path is discarded. If the same path is already contained in the routing table, only the last experienced one is kept.

Example 3.2.1 *Let $p_{exp1} = [v1 \rightarrow v2 \rightarrow v3 \rightarrow v5]$ be the experienced path of a bee. When the bee returns to its hive v_1, the following paths are evaluated for waggle-dance advertising: $p_1 = [v_1 \rightarrow v_2]$, $p_2 = [v_1 \rightarrow v_2 \rightarrow v_3]$, $p_3 = [v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_5]$.*

Algorithm 3.3: BeeNet ConstructSolution procedure

```
   Input  : bee, current node i
   Output : bee, path experienced by bee
 1 procedure ConstructSolution (bee, i);
 2 if bee has not reached bee.destination then
 3 |   if bee.type == follower then
 4 |   |   if ∃neighbor(i) ∈bee.preferredPath then
 5 |   |   |   bee.nextHop = applyFollowerTransitionRule();
 6 |   |   else
 7 |   |   |   bee.nextHop = applyForagerTransitionRule();
 8 |   |   end
 9 |   else
10 |   |   bee.nextHop = applyForagerTransitionRule();
11 |   end
12 |   if loop detected in bee.experiencedPath then
13 |   |   bee.experiencedPath.deleteLoop(i);
14 |   else
15 |   |   bee.experiencedPath.add(i, (Time.current()−bee.lastHopTime));
16 |   end
17 |   bee.lastHop ← i;
18 |   bee.lastHopTime ←Time.current();
19 |   MoveToNode(bee.nextHop);
20 |   ConstructSolution(bee, bee.nextHop);
21 else
22 |   ReturnToSource();
23 |   PerformWaggleDance(bee, bee.source);
24 end
```

3.2.6 Forwarding of data packets

When a data packet is sent from a source node s to a specified destination r, all paths to r, known at s, are extracted from the routing table. The path on which the data packet is sent is based on a roulette wheel selection [42]:

$$P_{forward}(path) = \frac{f_{path}}{f_{colony}(r)} \tag{3.8}$$

The reason for not only using the most optimal path with the highest fitness value for forwarding is to establish a simple form of fitness proportional load balancing. If always the same path is chosen for a destination, the path may becomes congested quickly and gets worse drastically, especially when the data packet traffic to the destination increases. However, since the probability of a path to be selected is based proportionally to its fitness value, it is ensured that the best paths are chosen more frequently than worse

Algorithm 3.4: BeeNet PerformWaggleDance procedure

```
   Input  : bee, source node s
   Output : updated routing table RT at s
 1 procedure PerformWaggleDance (bee, s);
 2 RT = s.getRT();
 3 expPath = bee.experiencedPath;
 4 forall (sub-)paths to
 5 (expPath.destination and all intermediate nodes) ∈ expPath do
 6 |   colonyFitness = calculateColonyFitness(path.destination);
 7 |   advertisePath = evaluatePath(pathToEvaluate , colonyFitness);
 8 |   if advertisePath == true then
 9 |   |   RT.get(path.destination).add(path);
10 |   else
11 |   |   Discard(path);
12 |   end
13 end
```

paths. Furthermore, if the fitness value of known paths only varies very marginally, it is avoided that only one of the practically equally good paths is used.

If no path is known for a data packets destination, it is forwarded to a random neighbor until a node is found, on which a path to the data packets destination is known. However, the last hop node of the data packet is ignored in that case, unless it is the only neighbor of the current node. The forwarding procedure is described in pseudocode 3.5.

Algorithm 3.5: BeeNets procedure for forwarding data packets

```
   Input  : current node i, data packet
   Output : forwarded data packet
 1 procedure FowardDataPacket (i, packet);
 2 if i == packet.destination then
 3 |   return;
 4 end
 5 RT = i.getRT();
 6 dest ← packet.destination;
 7 if i == packet.source or packet.path not set then
 8 |   if RT.get(dest) is not empty then
 9 |   |   colonyFitness = calculateColonyFitness(dest);
10 |   |   chosenPath = selectPath(RT.getPaths(dest), colonyFitness);
11 |   |   packet.path ← chosenPath;
12 |   else
13 |   |   select randomNeighbor of i ≠ packet.lastHop;
14 |   |   ForwardDataPacket(randomNeighbor, packet);
15 |   end
16 else
17 |   ForwardDataPacket(packet.path.getNextHop(), packet);
18 end
```

3.3 SlimeMoldNet

Šešum-Cavic et al. propose a swarm intelligent algorithm for P2P lookup in [40] that mimics the behavior of the Dictyostelium discoideum slime mold in its life-cycle. It is based on the Dictyostelium discoideum numerical optimization algorithm presented in [33]. The adaption of the Dictyostelium discoideum optimization algorithm for routing in unstructured P2P networks, called SlimeMoldNet (SMNet), presented in this section is based on those two works.

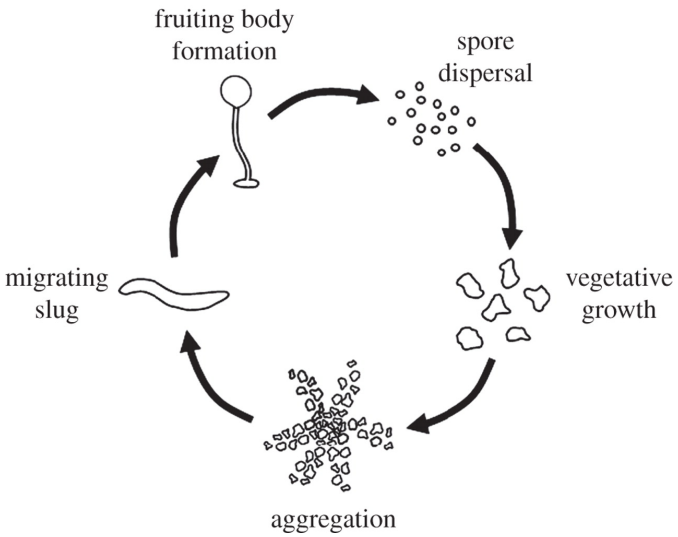

Figure 3.6: Life-cycle of the Dictyostelium discoideum slime mold [20]

.

The slime mold Dictyostelium discoideum (Dd) is a social collective of self-organizing amoebas that goes through a life-cycle visualized in figure 3.6. The goal of Dd amoebas is their feeding and therefore a constant supply of food that consists of bacteria and decaying material in the soil. The Dd life-cycle has several stages: vegetative movement, mound aggregation, slug movement, fruiting body formation and spore dispersal [24].

In the first stage, vegetative movement, amoebas navigate through the soil on their own, using tentacle-like pseudopods, and search for food. Furthermore, in this state of adequate nourishment, amoebas may procreate by fission. An amoeba stays in this state until the food supply shrinks. If not enough food supply is available, an amoeba begins to starve [24].

At this point, the collaboration of amoebas begins and the aggregation stage of the life cycle starts. Amoebas in this stage communicate indirectly by emitting a pheromone called cyclic Adenosine Monophosphate, the so-called cAMP. More specifically, one of the amoebas takes the role of the Pacemaker which releases cAMP and therefore causes all others to be drawn to it until a mound is formed [24].

When aggregating to a mound, the amoebas emit a slimy substance. In the mound, amoebas start to organizes themselves into two categories, prespore and prestalk, based on their fitness level. The fitness level is determined on an amoebas level of nourishment and therefore on how efficient it was when collecting food in the vegetative stage. Prespore amoebas move to one end of the mound and form the head, while the prestalk form the slime molds tail [24].

At the end of this aggregation process, the mound of amoebas have formed a slug and starts moving to a source of light. Movement of this collective is directed by head amoebas through emitting cAMP. The amoebas try to reach the surface of the soil until they die [24].

If they succeed to reach the surface, amoebas start organizing themselves into a fruiting body. The least fit amoebas, which formed the tail of the mound, sacrifice themselves and die after forming the stalk of the fruiting body. After that, the head amoebas climb the stalk and transform into spores. Spores are dispersed by environmental factors such as animals or the wind. After the dispersal and the process of germination, spores become active amoebas again and therefore the life-cycle begins again at the vegetative stage [24].

3.3.1 SlimeMoldNet algorithm for Routing in Unstructured P2P Networks

In the domain of routing in unstructured P2P networks, the network is considered as a landscape of soil. The soil contains concentrated areas of food, the P2P network nodes, on which Dd amoebas live. Each P2P node contains another kind of bacteria and therefore a different specific type of food.

At a node, amoebas procreate after the configurable `amoebaSpawnInterval` has passed. Therefore, after this interval has passed, always the same `amountOfSpawnedAmoebas` is created at a P2P node. Amoebas try to gather food from a specific food source in the soil and therefore try to reach a specified node in the P2P network.

Therefore, in SMNet, there exists an important restriction. A generation of amoebas, spawned by procreation at a P2P node at a point in time, is only able to digest a single type of food (bacteria). Only at a single node in the network, known at the node where the amobeas are spawned, this type of food is available. Which type of food the newly created amoebas are able to process, and therefore their target node, is chosen randomly.

As discussed, the survival or death of amoebas in the process of forming the fruiting body depends on their fitness level. The fitness level relies on how much food amoebas were able to find during the vegetative state [24].
Thus, the goal of an amoeba can be interpreted as to maximize their fitness and therefore

to optimize foraging. In the laid out scenario for P2P networks, this would mean to optimize the path to the node, where the food is available.

The problem of path optimization can be interpreted as a single numerical optimization problem. In [33], the goal of the single numerical optimization algorithm is specified as to optimize a function f based on decision variables $x_1, x_2, ..., x_n$ that are limited to a specific domain. The values of the function are defined as decision space [33].
For the domain of routing, an amoeba strives to maximize the fitness function of paths to food sources:

$$\begin{aligned} f_{max} &= f({x_1}^{min}, {x_2}^{min}, ..., {x_n}^{min}) \quad \text{such that} \\ f_{max} &> f(x_1, x_2, ..., x_n), \forall \{x_1, x_2, ..., x_n\} \end{aligned} \tag{3.9}$$

More specifically, the fitness function of a path to a food source is calculated by the same formula as presented for BeeNet in equation 3.2:

$$f_{path} = \left(\frac{1}{H_{path}}\right)\left(\frac{1}{D_{path}}\right) \tag{3.10}$$

where H_{path} is the amount of hops taken on the path from a source node to a destination node and D_{path} is the accumulated delay experienced on this path.

At each node of the P2P network, a routing table is held. For each known destination node in the network, paths, their accumulated delay and their corresponding fitness value are stored. The set of known destinations is a subset of all participating nodes in the P2P network, is predetermined and may also change during run time. The exact method of predetermination of known destinations depends on the P2P application (e.g. list of user names in the contact list of a P2P messanger or a list of known unique P2P node Ids). The class diagram of an routing table element is shown in figure 3.7.

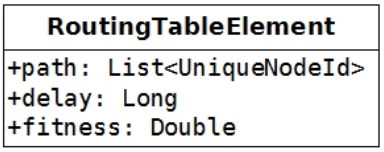

Figure 3.7: Element of a SMNet routing table

The overall model of the algorithm, shown in pseudocode 3.6, is similar to those presented in [33] and [23]. Each spawned amoeba goes through the stages vegetative movement, aggregation, mound forming, slug movement and dispersal of the Dd life-cycle. The procedures for each life-cycle stage are described in detail in the following subsections.

3.3.2 Initialization

After the `amoebaSpawnInterval` has passed, the creation of the configurable `amountOfSpawnedAmoebas` and their initialization is carried out at their source node.

Algorithm 3.6: SlimeMoldNet, a Dd based routing algorithm

Input : source node s, food sources R
Output : optimal paths from s to all food sources $r \in R$

```
1  procedure Dd (s, R);
2  while not terminated do
3      if amoebaSpawnInterval has passed then
4          spawnedAmoebas = spawnAmoebas(amountOfSpawnedAmoebas);
5          Initialize(spawnedAmoebas);
6          forall amoebas ∈ spawnedAmoebas do
7              VegetativeMovement;
8              Aggregation;
9              Mound;
10             SlugMovement;
11             Disperse;
12         end
13     end
14 end
```

Amoebas are represented by intelligent software agents, shown in figure 3.8. In this routing

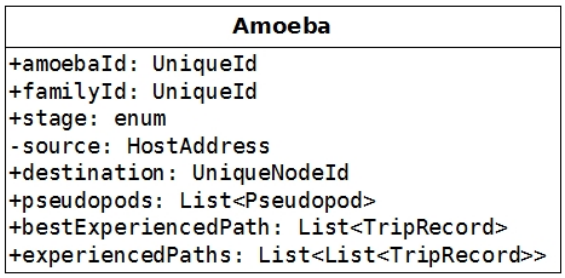

Figure 3.8: Class diagram of a SMNet amoeba

algorithm, amoebas spawned at the same node and at the same time are considered to be from the same "amoeba family", shown in figure 3.9. An additional constraint for SMNet is, that a family of spawned amoebas always has the same source node and the same destination node (specific food source). An amoeba searches not only for the specific type of food it is able to digest and which is only available at a single known P2P node in the network, it focuses on the search for the best path to this node. The best path between two nodes is considered to be the fastest one timewise (experienced delay).
Furthermore, these amoebas can only transition together to the aggregation stage of the Dd life-cycle. Thus, they have a shared `mealCount` and have, in addition to their own unique identifier, an unique family identifier. The `mealCount` is a counter that is increased every time the best known path to the destination is improved.

In SMNet, there exist three types of software agents. First, as mentioned, amoebas. Furthermore, amoebas spawn a primitive second software agent type in the vegetative stage: pseudopods. Moreover, amoebas are able to aggregate and therefore, technically, a third type of software agent is built: a slug. It is important to note that amoebas know the host address of their source. Therefore, spawned pseudopods and the built slug, can return directly to the their source node in a P2P way.

Like in [23], an amoeba is exactly in one stage of the Dd life-cycle at all times: "VEGETATIVE", "AGGREGATION", "MOUND", "SLUG" or "DISPERSAL". Amoebas always start in the vegetative movement stage. During the life-cycle, an amoeba memorizes all experienced paths to its destination.

The pseudocode of this procedure is shown in algorithm 3.7.
First, a new `family` of amoebas is created. Its unique id is generated using the `generateUniqueFamilyId` method. Furthermore, the `mealCount` and the amount of search steps `searchTime` is initialized to 0. Additionally, a random destination for the whole amoeba family is selected by calling the `selectRandomDestination` method.
After the amoeba family, all of its amoebas are initialized. For each amoeba a unique id is generated by the `generateUniqueAmoebaId` method. After setting the amoebas `familyId`, the amoebas source and the selected family destination, the state of the amoeba is set to "VEGETATIVE".

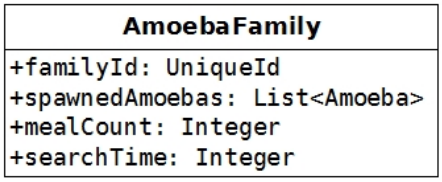

Figure 3.9: Class diagram of a SMNet amoeba family

3.3.3 Vegetative Movement

After initialization of an amoeba family, the P2P network is foraged using pseudopods (*pp*), represented by software agents shown in figure 3.10. Pseudopods are spawned by amoebas of a family iteratively and at the same time. After each vegetative step it is checked, if the whole amoeba family transitions to the next stage of the Dd life-cycle.

The amount of spawned pseudopods is constrained by a configurable limit `maxPseudopodLimit`. It is important to note, that the limit applies to the whole family. Thus, an amoeba family must not spawn more than `maxPseudopodLimit` pseudopods at a vegetative step. Pseudopods try to find paths to the amoebas destination. However, they are only allowed to move in the P2P network for a maximum amount of

Algorithm 3.7: Initialization of SMNet amoebas

Input : source node s, possible $destinationNodes$, $spawnedAmoebas$
Output : initialized family of spawned amoebas

```
1  procedure Initialize (s, destinationNodes, spawnedAmoebas);
2  family = new AmoebaFamily();
3  familyId = generateUniqueFamilyId();
4  family.familyId ← familyId;
5  family.spawnedAmoebas ← spawnedAmoebas;
6  family.mealCount ← 0;
7  family.searchTime ← 0;
8  familyDestination = selectRandomDestination(destinationNodes);
9  forall amoebas ∈ spawnedAmoebas do
10 |  amoebaId = generateUniqueAmoebaId();
11 |  amoeba.amoebaId ← amoebaId;
12 |  amoeba.familyId ← familyId;
13 |  amoeba.source ← s;
14 |  amoeba.destination ← familyDestination;
15 |  amoeba.state ← VEGETATIVE;
16 end
```

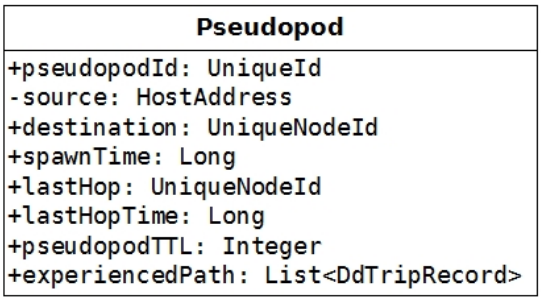

Figure 3.10: Class diagram of a SMNet pseudopod

`pseudopodMaxTTL` hops before they are terminated [23]. The `maxPseudopodLimit` and the `pseudopodMaxTTL` ensure that the P2P overlay network is not congested by too much routing related traffic and improve the algorithms scalability. If a pseudopod experiences a loop, and therefore visits a node on its path twice, it erases all information of the loop from its memory.

As in [23], pseudopods travel the network by choosing the next hop randomly, like random walkers [30]. However, in SMNet, there is the additional constraint, that the last hop node of a pseudopod is only chosen as next hop if it is the only neighbor of the current node. In that case, the probability for a pseudopod to choose a neighbor n as next hop

at the current node c is:

$$P(n)_{pp} = \frac{1}{|N_c|} \tag{3.11}$$

where N_c is the set of neighbors of the current node c excluding the pseudopods last hop node. If a pseudopod reaches its destination, it returns directly to its source amoeba (and therefore source node) in a P2P way.

The spawning of pseudopods at each vegetative step (iteration) is shown in pseudocode 3.8.
Until the `maxPseudopodLimit` is reached, pseudopods are spawned sequentially by all amoebas of a family. In order to determine the `index` of the amoeba which spawns the next pseudopod, a `counter` is incremented after each loop pass. Before a pseudopod starts its movement procedure `PseudopodMovement`, all of its attributes are initialized. A new unique id is generated using the `generateUniquePseudopodId` method, the pseudopods source, destination and `lastHop` are set. Furthermore, the pseudopods `spawnTime` and the `lastHopTime` are set to the current time, determined by method `Time.now`. Lastly, the pseudopods `TTL` is set to 0 and the source node is added to the `experiencedPath`.

Algorithm 3.8: SMNet: Spawning of pseudopods

```
   Input : source node s, amoeba family amoebaList
 1 procedure SpawnPseudopods (s, amoebaList);
 2 counter = 0;
 3 while counter < maxPseudopodLimit do
 4 |   index = counter mod amountOfSpawnedAmoebas;
 5 |   amoeba = amoebaList.get(index);
 6 |   pod = new Pseudopod();
 7 |   pod.pseudopodId = generateUniquePseudopodId();
 8 |   pod.amoebaId ← amoeba.amoebaId;
 9 |   pod.source ← s;
10 |   pod.destination ← amoeba.destination;
11 |   pod.spawnTime ←Time.now();
12 |   pod.lastHopTime ←Time.now();
13 |   pod.lastHop ← s;
14 |   pod.pseudopodTTL ← 0;
15 |   pod.experiencedPath(s, 0);
16 |   PseudopodMovement(s, pod);
17 |   counter++;
18 end
```

Algorithm 3.9: SMNet: Movement of a pseudopod

```
   Input : current node c, pseudopod pod
 1 procedure PseudopodMovement (c, pod);
 2 if c == pod.destination then
 3 |   pod.experiencedPath.add(c, (Time.current()−pod.lastHopTime));
 4 |   ReturnToAmoeba(pod.source);
 5 else
 6 |   if pod.pseudopodTTL== pseudopodMaxTTL then
 7 |   |   kill(pod);
 8 |   else
 9 |   |   nextHop = selectRandomNeighbor(c, pod.lastHop);
10 |   |   if loop detected in pod.experiencedPath then
11 |   |   |   pod.experiencedPath.deleteLoop(c);
12 |   |   else
13 |   |   |   pod.experiencedPath.add(c, (Time.current()−pod.lastHopTime));
14 |   |   end
15 |   |   pod.pseudopodTTL++;
16 |   |   pod.lastHop ← c;
17 |   |   pod.lastHopTime ←Time.current();
18 |   |   MoveToNode(nextHop);
19 |   |   PseudopodMovement(nextHop, pod);
20 |   end
21 end
```

The pseudopod movement is described in detail in pseudocode 3.9 as follows.
If a pseudopod has reached its destination, it adds the destination node and the delay experienced since the last hop node to its `experiencedPath` memory and returns to its amoeba by calling the `ReturnToAmoeba` method.
Otherwise, if the `pseudopodMaxTTL` is reached, the pseudopod is terminated. If that is not the case, the next hop node of the pseudopod is determined. It is chosen randomly by using the `selectRandomNeighbor` method. The method returns the pseudopods last hop, if it is the only neighbor of the current node. Otherwise, the last hop node is ignored by the `selectRandomNeighbor` method.
Before incrementing the pseudopods `TTL` by 1, setting the `lastHop` as the current node and storing the current time in the pseudopods `lastHopTime`, the current node in addition to the experienced delay since the last hop node are added to the `experiencedPath`. Then the pseudopod is transfered to the next hop through execution of the `MoveToNode` method and the `PseudopodMovement` procedure is executed again.

When all pseudopods have returned or a configurable `pseudopodWaitTime` has been surpassed, the state of the amoebas of the family is determined. If the amoebas show no

Algorithm 3.10: SMNet: Vegetative Movement

```
   Input : source node s, amoeba family amoebaFamily
 1 procedure VegetativeMovement (s, amoebaFamily);
 2 mealCountIncreased ← false;
 3 family.searchTime++;
 4 SpawnPseudopods(s, amoebaFamily.spawnedAmoebas);
 5 WaitForPseudopods(amoebaFamily.spawnedAmoebas, pseudopodWaitTime);
 6 forall pod ∈ returnedPseudopods do
 7 |   amoeba = family.getAmoeba(pod.amoebaId);
 8 |   if calculateFitness(pod.experiencedPath) >
 9 |     calculateFitness(amoebaFamily.getBestPath()) then
10 |     amoeba.bestExperiencedPath ← pod.experiencedPath);
11 |     amoeba.experiencedPaths.add(pod.experiencedPath);
12 |     if mealCountIncreased == false then
13 |       family.mealCount++;
14 |       mealCountIncreased ← true;
15 |     end
16 |   else
17 |     amoeba.experiencedPaths.add(pod.experiencedPath);
18 |   end
19 end
20 if amoebaFamily.isStarving() then
21 |   Aggregation(family.spawnedAmoebas);
22 else
23 |   VegetativeMovement(s, amoebaFamily);
24 end
```

improvement of found paths to the destination over multiple search steps and therefore the `mealCount` stagnates, the amoebas are considered to starve. Starving causes the amoebas to transition to the aggregative stage of the Dd life-cycle. More specifically, in order for an amoeba family to be considered starving, two conditions have to be met. First, the following probabilistic check is done [23], [33]:

$$p(x < \frac{localSearchTime - mealCount}{localSearchTime}) \tag{3.12}$$

where `localSearchTime` is the number of vegetative steps made by the family and $x \in (0, 1]$ is a random value. Thus, if the `mealCount` stagnates and the amoeba family starves, the chance to transition increases. Furthermore, a second transition condition has to be met [23], [33]:

$$localSearchTime > minSearchTime \tag{3.13}$$

where `minSearchTime` is the minimum amount of vegetative steps that have to be made before amoebas are allowed to transition.
If both transition conditions are fulfilled, all amoebas of the family transition to the next stage of the Dd life-cycle. Otherwise, they enter a new vegetative step.

The procedure at each vegetative step is described detailly in pseudocode 3.10.
First, for a vegetative step, the indicator, if the `mealCount` has been increased in this iteration (`mealCountIncreased`), is initialized to boolean $false$. Furthermore, at the beginning of the vegetative step, the `searchTime` is increased by 1. Next, the pseudopods are spawned as described in pseudocode 3.8.

Then, the procedure halts, by calling the `WaitForPseudopod` method, until the `pseudopodWaitTime` has passed or all spawned pseudopods have returned. When the procedure resumes, it is checked if the `experiencedPath` of the pseudopod improves the amoebas personal best and the families personal best. The fitness values are calculated by using the `calculateFitness` method. If the families personal best is improved and the `mealCount` has not been increased at the current vegetative step, the `mealCount` is increased by 1. Additionally, the indicator `mealCountIncreased` is set to boolean *true* for this iteration.

Lastly, it is checked if the amoebas of the family are starving by calling the `isStarving` method. This method checks if conditions 3.12 and 3.12 are met. If that is the case, the `Aggregation` procedure is executed. Otherwise, the amoebas enter a new vegetative step by executing the `VegetativeMovement` procedure again.

3.3.4 Aggregation

At this stage, one of the families amoebas becomes a pacemaker and starts releasing cAMP. How Dd amoebas determine if the take the pacemaker role in nature, is not known [24]. Moreover, since all amoebas stay at their source node in the vegetative state, the determination of the pacemaker amoeba can be arbitrary. Therefore, in SMNet, the pacemaker is chosen uniformly random.

Indirect communication through cAMP release causes the other amoebas of the family to get drawn to the pacemaker. When reaching the pacemaker, they start forming a mound [24]. The mounds structure is shown in figure 3.11.

Figure 3.11: Class diagram of a SMNet mound

Since, all amoebas of a family already are at the same node, the procedure shown in

pseudocode 3.11 is simple. After the creation of the mound structure, for all aggregating amoebas, the state "AGGREGATION" is set. Moreover, each amoeba is added to the mounds `amoebaList`.

Algorithm 3.11: SMNet: Aggregating to a mound

```
1 d Input  : source node s, amoebaList
  Output : slug
2 procedure Aggregation(s, amoebaList);
3 mound = new Mound();
4 forall amoebas ∈ amoebaList do
5 |   amoeba.state ← AGGREGATION;
6 |   mound.amoebaList.add(amoeba);
7 end
8 FormSlug(s, mound);
```

3.3.5 Mound

After the mound is formed, the Mound stage of the algorithm is automatically entered. In this phase, the amoebas contained in the mound, start organizing themselves [24]. The amoeba with the best experienced path, and therefore the highest fitness, becomes the head of the mound. All other amoebas become its tail. For the evaluation the fitness function 3.10 is used.

After this process, the amoebas have organized themselves to a slug structure. Logically, the slug is an aggregation of software agents controlled by the head amoeba, while the amoeba software agents forming the slugs tail are inactive. However, technically, the slug is a separate software agent as described in 3.12.

This self-organization process is described in pseudocode 3.12.
The fitness value of the personal best of each amoeba contained in the mound structure is calculated and compared to the fitness value of the current head amoeba using the `calculateFitness` method.
If no head amoeba exists yet or the current amoeba has a better fitness value than the current head amoeba, it is set as the new head amoeba. Otherwise, the current amoeba becomes part of the slugs tail. Additionally, each amoebas state is updated to "SLUG". After the slugs source and destination is set using the head amoeba, it starts its movement as described in pseudocode 3.13.

3.3.6 Slug Movement

After amoebas have organized themselves in the slug structure, the slug starts moving in the direction of its destination. Therefore, the slug is indirectly steered by the head

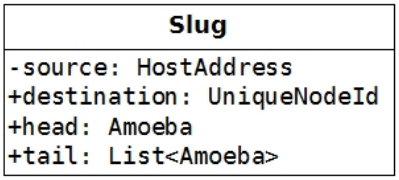

Figure 3.12: Class diagram of a SMNet slug

Algorithm 3.12: SMNet: Forming a slug from the mound

```
   Input  : source node s, mound
   Output : mound
 1 procedure FormSlug(s, mound);
 2 slug = new Slug();
 3 forall amoebas ∈ mound do
 4 |   amoeba.state ← SLUG;
 5 |   if slug.head is empty OR
 6 |   calculateFitness(amoeba.bestExperiencedPath) >
 7 |   calculateFitness(slug.head.bestExperiencedPath) then
 8 |   |   slug.tail.add(slug.head);
 9 |   |   slug.head ←amoeba;
10 |   else
11 |   |   slug.tail.add(amoeba);
12 |   end
13 end
14 slug.source ← slug.head.source;
15 slug.destination ← slug.head.destination;
16 SlugMovement(s, slug);
```

amoebas released cAMP [24].

The slug of amoebas follows the best experienced path of the head amoeba to the destination. On each intermediate node, the routing table is updated by the slug. More specifically, all sub-paths from the current node to the slugs source and destination and to all intermediate nodes are added to the routing table. If a path already exists in the routing table, it is replaced.

Furthermore, the fitness function values of known paths to the slugs destination, held by the current nodes routing table, are compared to the fitness function value of the slugs remaining sub-path (path from the current node to the slugs destination). If a known path to the destination at the current node is better than the remaining part of the head amoebas personal best, the amoeba updates the personal best accordingly.

Example 3.3.1 *Let $p_{best} = [v1 \rightarrow v2 \rightarrow v3 \rightarrow v4 \rightarrow v5]$ be the best experienced path by a slugs head amoeba. Assume the head amoebas best experienced path was not improved at node v_1 and v_2. When reaching node v_3, the following paths are added to its routing table: $p_1 = [v_3 \rightarrow v_2]$, $p_2 = [v_3 \rightarrow v_2 \rightarrow v_1]$, $p_3 = [v_3 \rightarrow v_4]$, $p_4 = [v_3 \rightarrow v_4 \rightarrow v_5]$. The best known path at v_3 to node v_5 is $p_{node} = [v_3 \rightarrow v_5]$. Assume the fitness value of p_{node} is higher than $p_{best}^{sub} = [v_3 \rightarrow v_4 \rightarrow v_5]$. The head amoebas p_{best} is then updated to $p'_{best} = [v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_5]$. After the update of p_{best}, the next hop of the slug is v_5.*

The movement procedure is shown in pseudocode 3.13 as follows:
Starting at the slugs source node, the currently best experienced path of the slugs head amoeba (`slug.head.`*bestExperiencedPath*) is followed. Therefore, the `index` of the current node on this path is calculated by executing the `indexOf` method. The next hop node on the path has `index+1` on the currently followed path. If the slug has not reached its destination, it moves to the next node on its path and the `SlugMovement` procedure is executed again.

At each intermediate node on the followed path, the routing table is updated as showed in example 3.3.1: All sub-paths from the current node to the slugs source, destination and to all other intermediate nodes are determined from the head amoebas currently best experienced path and added to the current nodes routing table. The routing table of the current node is accessed using the `getRT` method.

Next, the slug tries to improve its own personal best. Therefore, the path with the highest fitness value, at the current node to the slugs destination `bestNodePath` is selected in the `getBestPath` method. Additionally, the sub-path from the current node c to the slugs destination (`slugSubPath`) is extracted from the currently best experienced path (which is currently followed). For both of these paths, the fitness value is calculated using the `calculateFitness` method. If the fitness value of the `bestNodePath` is higher than the `slugSubPath`, the current personal best and therefore the path followed by the slug is updated. In the `updatePath` method, of the current `bestExperiencedPath`, the fraction stated by the sub-path `slugSubPath` is replaced with the `bestNodePath`. Furthermore, the improved best experienced path is added to the list of the head amoebas `experiencedPaths`. After the update, the slug now follows the new remaining `bestNodePath`.

When the slug reaches its destination, the `SlugMovement` procedure is aborted and the fruiting body is formed. The `BuildFruitingBody` procedure is described in pseudocode 3.14.

3.3.7 Fruiting Body And Dispersal

When the slug has reached its destination, a fruiting body is formed. Amoebas of the tail form the stalk of the fruiting body and die. Only the head amoeba climbs the stalk and transforms into a spore [24].

Algorithm 3.13: SMNet: Slug Movement

```
   Input  : current node c, slug
 1 procedure SlugMovement(s, slug);
 2 head = slug.head;
 3 if c == slug.destination then
 4 |   BuildFruitingBody(c, slug);
 5 |   return;
 6 end
 7 if c ≠ slug.source then
 8 |   forall sub-paths to (slug.source, slug.destination
 9 |           and to all intermediate nodes ≠ c) ∈ head.bestExperiencedPath do
10 |     c.getRT().get(path.destination).add(path);
11 |   end
12 |   bestNodePath = getBestPath(c.getRT().get(slug.destination));
13 |   slugSubPath = head.bestExperiencedPath.subPath(c, slug.destination);
14 |   if calculateFitness(bestNodePath) >
15 |     calculateFitness(slugSubPath) then
16 |     updatePath(head.bestExperiencedPath, bestNodePath);
17 |     slug.head.experiencedPaths.add(slug.head.bestExperiencedPath);
18 |   end
19 end
20 index = head.bestExperiencedPaths.indexOf(c);
21 MoveToNode(head.bestExperiencedPaths.get(index + 1));
22 SlugMovement(head.bestExperiencedPaths.get(index + 1), slug);
```

The SMNet procedure for building the fruiting body, shown in pseudocode 3.14, is a bit more sophisticated.
After the spore structure, shown in figure 3.13 is created, its source is set to the source of the slug. The best `bestExperiencedPath` of the head amoeba is automatically added to the spores `experiencedPaths` list.
For all other experienced paths of the head amoeba and for all experienced paths of tail amoebas, the fitness value is compared to the fitness value of the head amoeba using the `calculateFitness` method. Only if a paths fitness is at least 0,85 times the overall best experienced paths fitness value, it is added to the spores `experiencedPaths`. Furthermore, a path is only added if it is not already part of the spores `experiencedPaths` list. The reasons for choosing the value 0,85 is, that all paths with significantly worse fitness than the best experienced paths should be discarded. However, to enable load balancing, discussed in subsection 3.3.8, it is advantageous to consider path slightly worse than the best experienced path.
Thus, tail amoebas only chance to be part of the spore is to contain a path only slightly

worse than the best experienced path. All other tail amoebas die.

The dispersal procedure is shown in pseudocode 3.15.
If the spore has not reached the amoeba source, the `ReturnToSource` method is called to do so. Then, the `Disperse` procedure is executed again. Otherwise, the routing table of the source node is accessed using the `getRT` method. As shown in example 3.3.2, not only the path to the destination of an experienced paths is added to the routing table, but also the sub-paths to all intermediate nodes. If a path already exists in the routing table, it is replaced.

Example 3.3.2 *Let* $p_{exp1} = [v1 \rightarrow v2 \rightarrow v3 \rightarrow v4 \rightarrow v5]$ *and* $p_{exp2} = [v1 \rightarrow v2 \rightarrow v3 \rightarrow v5]$ *be the experienced paths of the spore. When the spore returns to* v_1*, the following paths are added to its routing table:* $p_1 = [v_1 \rightarrow v_2]$, $p_2 = [v_1 \rightarrow v_2 \rightarrow v_3]$, $p_3 = [v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4]$, $p_4 = [v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5]$, $p_5 = [v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_5]$.

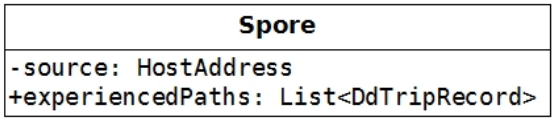

Figure 3.13: Class diagram of a SMNet spore

Algorithm 3.14: SMNet: Build Fruiting Body

```
   Input  : current node c, slug
1  procedure BuildFruitingBody (c, slug);
2  spore = new Spore();
3  spore.source ← slug.source;
4  bestPath = slug.head.bestExperiencedPath;
5  spore.experiencedPaths.add(bestPath);
6  forall amoebas ∈ slug.head and slug.tail do
7      forall paths ∈ amoeba.experiencedPaths do
8          if path ∉ spore.experiencedPaths &&
9          calculateFitness(path) > 0.85 calculateFitness(bestPath)
            then
10             spore.experiencedPaths.addPath(path);
11         end
12     end
13 end
14 Disperse(c, spore);
```

Algorithm 3.15: SMNet: Dispersal procedure

```
   Input  : current node c, spore
   Output : updated routing table RT
 1 procedure Disperse (c, spore);
 2 if c ≠ spore.source then
 3 |   ReturnToSource();
 4 |   Disperse(spore.source, spore);
 5 else
 6 |   RT = c.getRT();
 7 |   forall expPath ∈ spore.experiencedPaths do
 8 |   |   forall paths to expPath.destination and all intermediate nodes do
 9 |   |   |   RT.get(path.destination).add(path);
10 |   |   end
11 |   end
12 end
```

3.3.8 Forwarding of data packets

When a data packet is sent from a source node s to a specified destination r, all paths to r, known at s, are extracted from the routing table. The path on which the data packet is sent is based on a roulette wheel selection [42]:

$$P_{forward}(path) = \frac{f_{path}}{f_{node}(r)} \tag{3.14}$$

$$f_{colony}(r) = \frac{1}{n} \sum_{i=1}^{n} f_{r,i} \tag{3.15}$$

where f_{node}(r) equals the average fitness function value of all paths currently held by the routing table for a specific destination node r and n is the number of paths to r contained in the routing table [42].

The reason for not only using the most optimal path with the highest fitness value for forwarding is to establish a simple form of fitness proportional load balancing. If always the same path is chosen for a destination, the path may becomes congested quickly and gets worse drastically, especially when the data packet traffic to the destination increases. However, since the probability of a path to be selected is based proportionally to its fitness value, it is ensured that the best paths are chosen more frequently than worse paths. Furthermore, if the fitness value of known paths only varies very marginally, it is avoided that only one of the practically equally good paths is used.

If no path is known for a data packets destination, it is forwarded to a random neighbor until a node is found, on which a path to the data packets destination is known. However,

the last hop node of the data packet is ignored in that case, unless it is the only neighbor of the current node. The forwarding procedure is described in pseudocode 3.5.

Algorithm 3.16: SMNets procedure for forwarding data packets

```
   Input  : current node c, data packet
   Output : forwarded data packet
 1 procedure FowardDataPacket (c, packet);
 2 if c == packet.destination then
 3 |   return;
 4 end
 5 RT = c.getRT();
 6 dest ← packet.destination;
 7 if c == packet.source or packet.path not set then
 8 |   if RT.get(dest) is not empty then
 9 |   |   nodeFitness = calculateNodeFitness(dest);
10 |   |   chosenPath = selectPath(RT.getPaths(dest), nodeFitness);
11 |   |   packet.path ← chosenPath;
12 |   else
13 |   |   select randomNeighbor of c ≠ packet.lastHop;
14 |   |   ForwardDataPacket(randomNeighbor, packet);
15 |   end
16 else
17 |   ForwardDataPacket(packet.path.getNextHop(), packet);
18 end
```

3.4 Summary

At the beginning of this chapter, a model of routing in unstructured P2P networks is presented. It is how routing for this domain, in its abstracted form, is interpreted in this master thesis. Then, adaptions of two swarm based algorithms are presented. Both of them route data packets based on paths experienced by intelligent software agents. The first one, BeeNet, an adaption of Bee Algorithm [42] and Bee Colony Optimization [50], is based on the foraging behavior of bees. The second one, SlimeMoldNet, an adaption of the Slime Mold algorithm [33] and based one of its adoptions for P2P lookup [23], mimicks the life-cycle of the Dictyostelium discoideum slime mold. The algorithms are evaluated and analyzed in chapter 6.

CHAPTER 4

Peer Model Framework Architecture

In this chapter the architecture of the frameworks pattern, its components and their composition is described. The general idea of the patterns structure is influenced by the modular network layer for sensornets proposed in [16]. Wireless sensornets are wireless networks in which sensors send monitored data to a data sink [36]. The sensornet framework decomposes the routing layer in three major components: the Routing Engine (calculates and maintains routes over the network based on the abstract topology provided by the Routing Topology module), the Forwarding Engine (receives packets, queries the Routing Engine and takes action based on the answer), Routing Topology (communicates with Routing Topology modules of other network nodes in order to create and maintain an abstract topology of the network). Analogously, the Peer Space framework uses three components with similar functionality: the Routing Decision Peer, the Forwarding Peer and the Routing Information Peer described in this chapter. However, while decomposing the network layer (and therefore routing) in meaningful components, the framework is created for another purpose than the fair benchmarking of routing algorithms, specifically to make the implementation of new routing protocols easier and to reuse code of existing ones.
The chapter is structured in the following way: After the description of the core pattern, its composition is discussed. Next, the rest of the frameworks components are introduced and the overall framework composition is defined. Then, the entry types used, the frameworks peers and their wirings are specified. The description of the frameworks services is not included in this chapter, but can be found in section 5.2. For a description of the wiring figures notation see subsection 2.1.

4.1 Peer Space Routing Pattern

Figure 4.1 shows the abstracted routing functionality of a P2P network node. The main routing task of network nodes is to forward data packets on routes from source to destination. Therefore, the neighbor(s) to which data packets should be sent on their route to the destination must be determined. Thus, routing decisions need to be made. These routing decisions might be based on collected and stored information (e.g. a routing table) or made ad-lib (e.g. forward data packets to randomly chosen neighbors). In order to collect routing relevant information, nodes in the P2P network may collaborate by exchanging routing relevant information via inter-node communication. The described

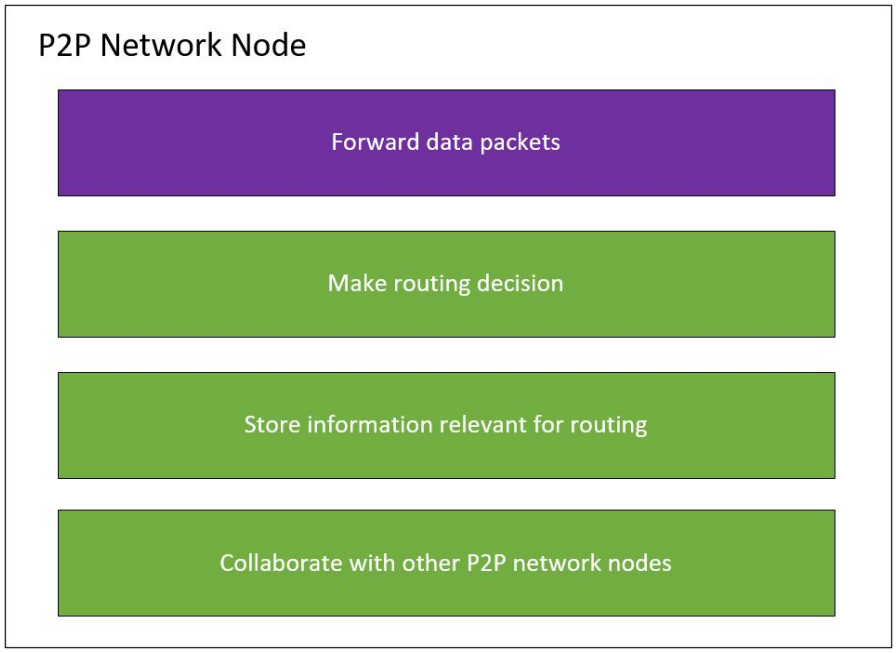

Figure 4.1: Routing functionalities of a P2P network node

routing functionality is mapped to the Peer Model domain as shown in figure 4.2. A node in a P2P network is embodied by a Node Peer that contains two sub peers: the Forwarding Peer (responsible for forwarding incoming data packets) and the Routing Peer (queried by the Forwarding Peer for the destinations to which data should be forwarded and responsible for storing routing relevant information). The main functionality of the Routing Peer is divided into two further sub peers: the Routing Decision Peer and the Routing Information Peer. Purple lines mark traffic regarding the forwarding of data, the green lines correspond to traffic regarding routing. Bold lines indicate traffic to other Node Peers. The pattern composition (how Node Peers communicate with each other) can be found in section 4.2, whereas the framework composition (how all peer types of the framework collaborate with each other) is defined in section 4.3. When determining how to process incoming data packets (in the form of a data entries in the

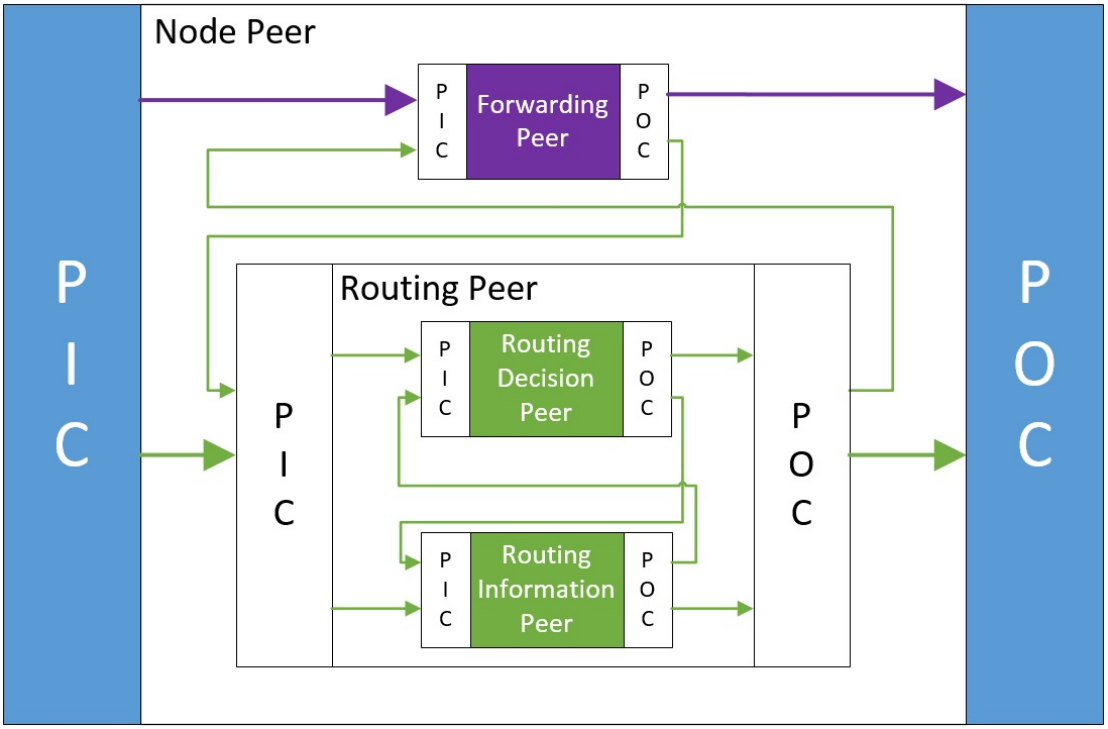

Figure 4.2: The architecture of a Node Peer in the Peer Space Routing Framework. Green arrows mark routing specific communication, violet arrows mark the forwarding of payload specific communication (data sent over the network).

Peer Space framework), the sub-components of the Node Peer collaborate as shown in figure 4.3. First, the Forwarding Peer sends a decision request to the Routing Decision Peer requesting a decision how to process a specific data entry. Generally, there are two possibilities: either the data entry should not be forwarded any further (when it has already reached its final destination at the current Node Peer, or when it should be dropped) or it is forwarded to at least one other Node Peer. The decision made by the Routing Decision Peer is obviously dependent on the routing algorithm. However, the Routing Decision Peer needs information held by the Routing Information Peer to be able to make an adequate routing decision and answer the decision request sent by the Forwarding Peer. Therefore, it sends an information request to the Routing Information Peer. The Routing Information Peer assembles the response based on the knowledge base held in its PIC (in the form of a routing information base entry). Note that, depending on the routing algorithm, the Routing Information Peer may not be able to assemble the request directly, but has to wait for an update of its knowledge base or has to request the information explicitly from other Node Peers. If that is the case, it generally implies some form of communication with Routing Information Peers of other Node Peers.

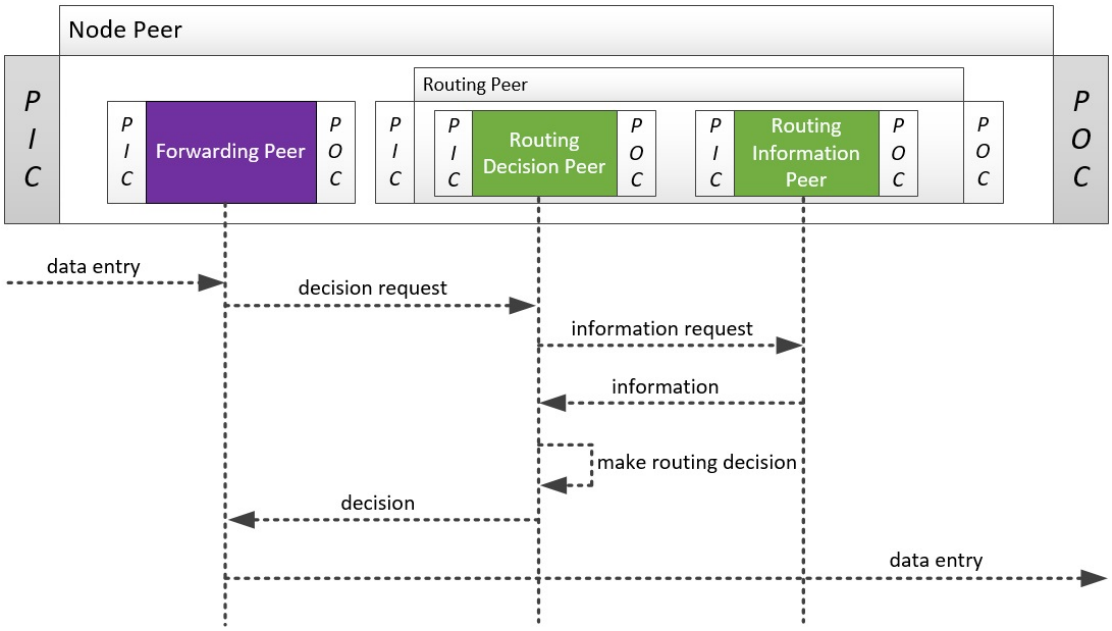

Figure 4.3: Simple sequence of the communication between sub-components of the Node Peer in order to determine how to forward a data entry.

4.2 Pattern composition

The basic pattern composition is rather simple. As shown in figure 4.4, Node Peers communicate with each other by simply putting entries in each other's PICs. The distribution to the respective sub peer, that handles the received entry, is then done internally by the Node Peers wirings. This pattern composition allows the creation of arbitrary network topologies.

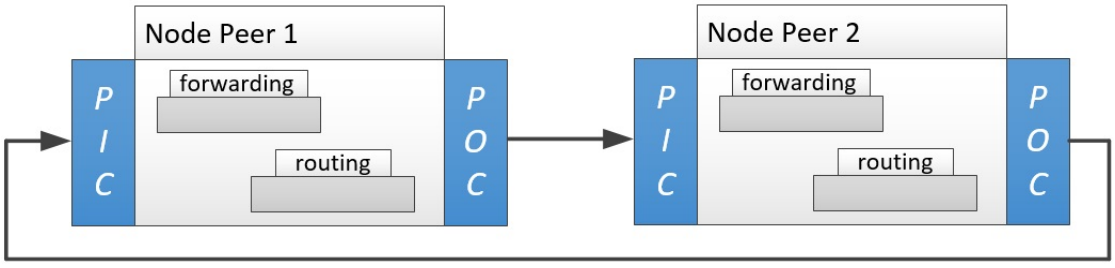

Figure 4.4: Basic pattern composition

However, in the Peer Space framework, the composition is more sophisticated. Although the concept of communication between Node Peers basically stays the same, the framework uses I/O Peers for relaying entries in inter-node-peer communication. I/O Peers decouple the communication specific concerns from the core pattern and enable additional features

like the simulation of transmission interference (e.g. transmission delay).
There exists exactly one I/O Peer per Node Peer Space, managing the communication of all Node Peers on that Peer Space. When a benchmark is run locally (i.e. all of the benchmarks Node Peers exist on the same Peer Space), their communication is managed by the same I/O Peer. Thus, there exists only one I/O Peer for the whole benchmark. When relaying entries for inter-node-peer communication locally on a Node Peer Space, the I/O Peer directly sends the entry to the target Node Peer. This configuration is shown in figure 4.5.

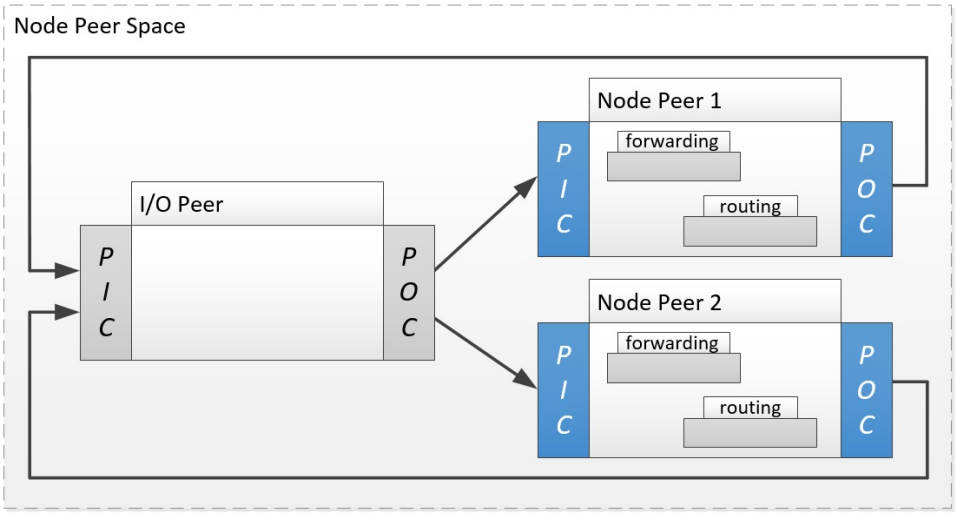

Figure 4.5: Local pattern composition in the Framework

If a distributed benchmark is run (i.e. there are multiple Node Peer Spaces containing Node Peers), the number of I/O Peers equals the number of the involved Node Peer Spaces. For the inter-node-peer communication between Node Peers on different Peer Spaces, the managing I/O Peer of the source Node Peer sends the entry to the managing I/O Peer of the target Node Peer. The target Node Peers I/O Peer then relays the entry to the target Node Peer. Furthermore, for inter-node-peer-space communication, entries are specifically marked such that the I/O Peer is able to differentiate between those locally sent and those received from Node Peers on other Node Peer Spaces. In the framework this is handled by using specific entry types for data entries (type “outD”) and routing communication entries (type “outR”) for that case. The distributed configuration is shown in figure 4.6.

4.3 Framework Composition

On the top-level of the framework, there are two types of Peer Spaces (the Control Peer Space and the Node Peer Space) and four types of Peers (Control Peer, Statistics Peer, Node Peer and I/O Peer). How the frameworks components collaborate is shown in figure 4.7.

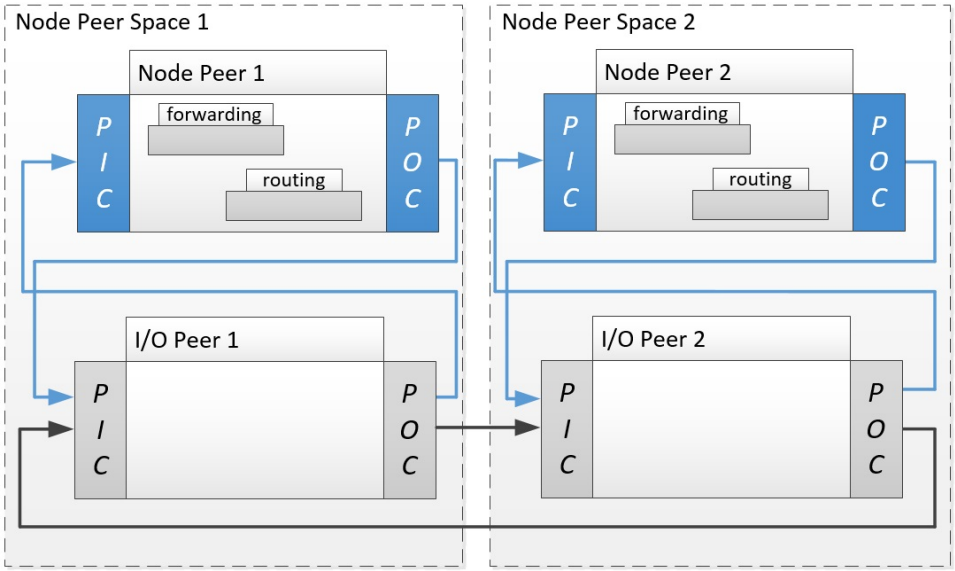

Figure 4.6: Distributed pattern composition in the Framework

There may exist multiple instances of the Node Peer Space in the routing framework (each possibly runs on an own physical computer instance). An instance of the Node Peer Space contains exactly one instance of the I/O Peer and at least one Node Peer instance. Node Peers embody servlets of the benchmarks Peer-to-Peer network and exchange information in the form of data entries (routed payload) and routing communication entries (routing algorithm specific communication between Node Peers). I/O Peers decouple the communication specific concerns from the core pattern by handling all inter-node-peer communication of their Node Peer Spaces Node Peers.
The Control Peer Space exists exactly once in the framework. It contains one instance of the Control Peer and the Statistics Peer. The Control Peer is the control center of the routing framework. It creates the topology of the benchmarks P2P network, initializes all other components accordingly, places data entries at random Node Peers, starts the benchmark, stops the benchmark and initiates the output of the benchmarks statistics. The Statistics Peer calculates the benchmarks statistics based on the data it receives from the networks Node Peers. The statistic relevant data is extracted from the coordination-data of entries used in inter-node-peer communication (in the form of time traces) and is always relayed by the Node Peers managing I/O Peer.
Figure 4.8 shows the sequence of a benchmark in the Peer Space Routing Framework. When a new Node Peer is spawned, it registers at the I/O Peer instance of its Node Peer Space. I/O Peers register at the Control Peer by informing it about their address and the addresses of all Node Peers managed by them. When the expected amount of Node Peers (defined in the benchmarks configuration) have registered at the Control Peer, the Control Peer builds the benchmarks network topology and initializes all other components based on that. After that, the initialization phase starts. In this phase, depending on

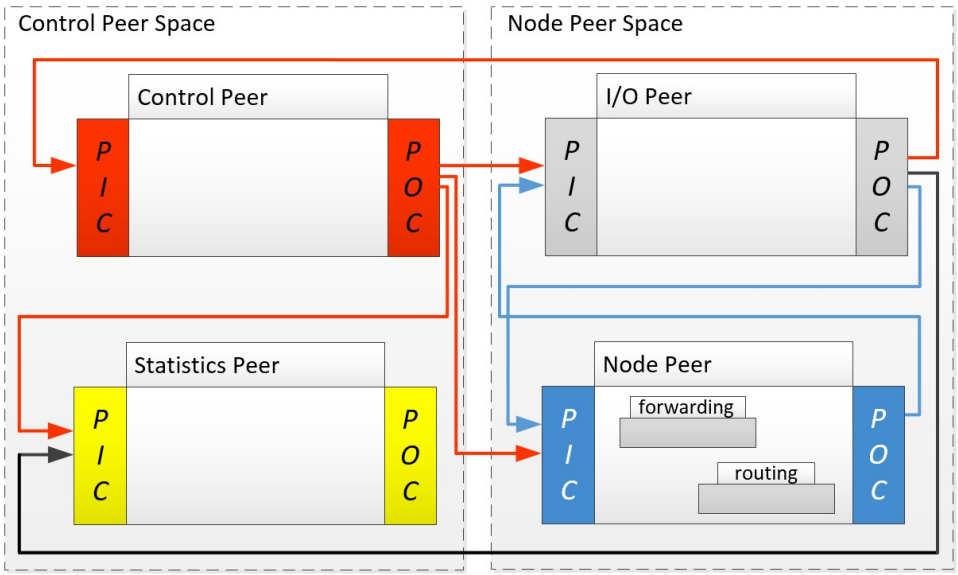

Figure 4.7: Composition of the Frameworks main components. The collaboration is shown in detail in figure 4.8.

the routing algorithm, Node Peers may build their knowledge base by communicate with each other (e.g. build a routing table). After the configurable time-frame of the initialization phase the Control Peer starts the benchmark. During the benchmark, data entries are forwarded from their source to their destination. Node Peers may also continue to exchange routing information to update their knowledge base. Furthermore, statistic relevant data, extracted from data entries and routing communication entries, which have reached their final destination or are dropped, are sent by I/O Peers to the Statistics Peer. The benchmark runs for a configurable time-frame. After this time-frame has passed, the Control Peer stops the benchmark and triggers the output of the calculated statistic.

4.4 Core Framework Components

In this section, the components of the framework are described in detail. The frameworks implementation specifics, as well as the description of its services and interfaces, is described in section 5.2.

4.4.1 Simplifications

For reasons of clarity and ease of understanding some simplifications are made.
It is important to note, that entries in the framework of types which are sent multiple times in a benchmark (e.g. entries of type “data” or “rtCom”) are not handled and sent separately by the framework, but handled in a container entry of the same type when certain conditions are met. Thus, entries of the same type, the same source, the

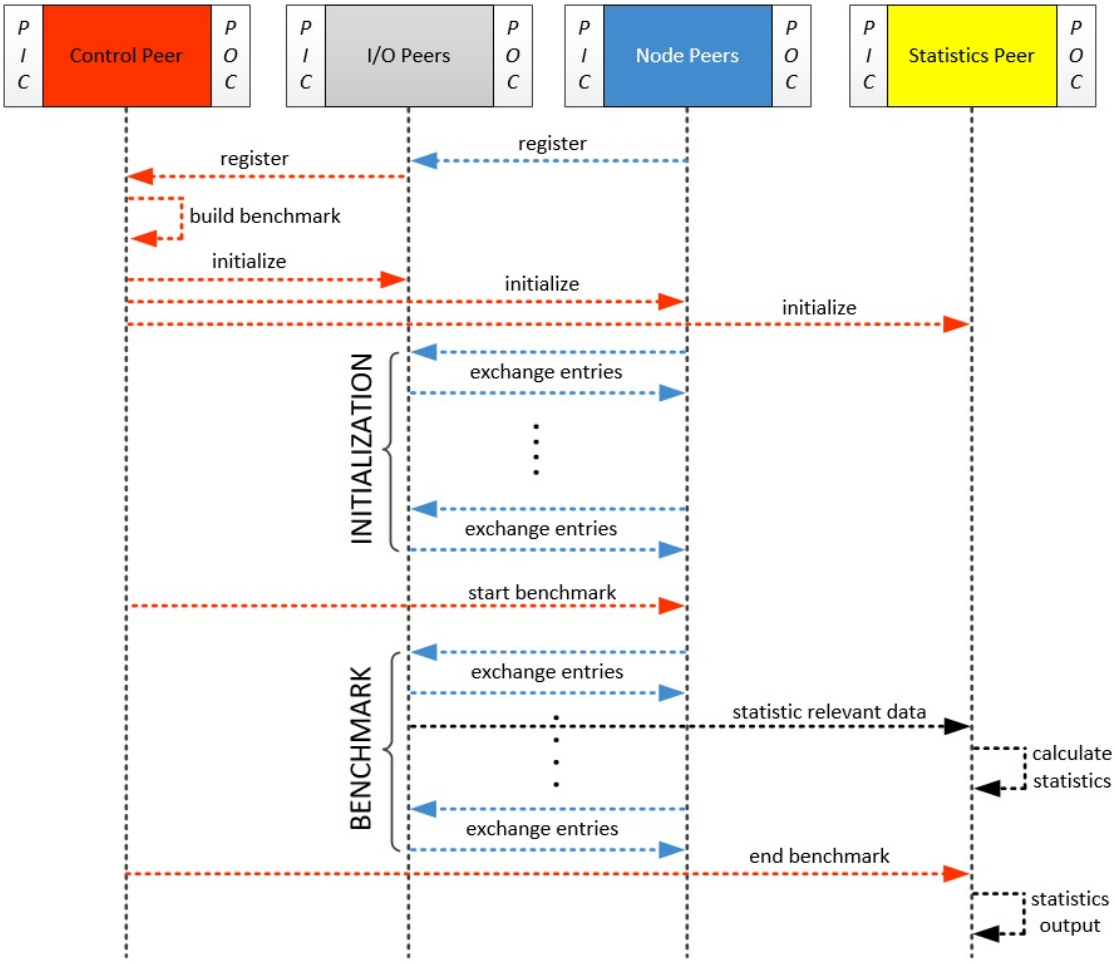

Figure 4.8: Sequence of a benchmark in the Routing Framework

same destination and with the same transmission interference (e.g delay) that are sent at the same time are merged into a single container entry which contains them in its coordination data. This is done due to performance optimization. Only one entry has to be handled by the Peer Model, while the affected services still consider the merged entries as separated. Otherwise, it would be technically impossible for the framework to handle a realistic amount of entries at the same time. This technique is also used internally in the frameworks peers and their sub peers in order to optimize the performance of processing entries. To ease the understanding of the internal processes of the framework, entries of the same type are always described and showed in the figures as processed separately in this section. However, in fact, instead of single entries, (multiple) container entries of the same type which might contain multiple single entries (but at least one), are processed.

Example 4.4.1 *Node Peer A sends 100 data entries to Node Peer B and 100 data entries to Node Peer C. The two sets of 100 entries are not sent separately but merged in*

a single container entry of type "data" each. This data entry contains the merged entries in its coordination data. When a container entry is received at Node Peer B or Node Peer C, the data entries are unpacked first and processed as single entries (in the same form as before the merge).

Additionally, since the proposed framework offers the possibility for algorithms to define a priority for the processing of data or routing related information, there are guards which are optionally and dynamically added at the benchmarks creation time (depending on the configured priority mode). This is applied everywhere in the framework where data and routing information competes for process time or resources i.e. dispatching, sending, relaying, or processing of entries. The dynamically added guards are of type "NONE". Thus, if the processing of data entries is prioritized, to all routing information related wirings which compete for processing time or resource a NONE-guard for entries of type "data" is added. Therefore, routing information related entries are only processed if no entry of type "data" is contained in the PIC of the peer. If the processing of routing information is prioritized, the same applies, but vice-versa. Due to reasons of simplicity, these optionally added guards are ignored in this section.

Example 4.4.2 *Assume the implemented routing algorithm prioritizes routing communication over the processing of data entries. In the Routing Information Peer, there is a wiring for updating the knowledge base of the routing algorithm at a Node Peer and one which is responsible for providing routing information to the Routing Decision Peer. Both of the wirings compete for the entry which contains the knowledge base. By adding a NONE-guard for entries of type "rtCom" to the latter wiring, the algorithm ensures that the knowledge base is always updated before providing routing information for decision making.*

Example 4.4.3 *Assume the implemented routing algorithm prioritizes the processing of data entries over processing of routing communication related entries. At the I/O Peer, entries are relayed from a sending Node Peer to a receiving Node Peer. By adding a NONE-guard for entries of type "data" to the wiring responsible for relaying routing communication entries at creation time, data entries are always relayed first.*

4.4.2 Entries

The following types of Entries are used in the proposed framework. In the brackets, their abbreviation in this chapters descriptions and figures are stated.

- **data entry (data):** Embodies the payload i.e. a data packet which should be routed from a source to a destination within the P2P network. The performance of data entries routing is benchmarked by the framework. In order to do that, a time trace of the path from source to destination is saved in a data entries co-data.
- **data wrapper entry (outD):** Wraps a data entry for sending between Peer Spaces.

- **decision request entry (dReq):** This type of entry is sent by the Forwarding Peer to the Routing Peer in order to request a routing decision for a specific data entry.
- **decision answer entry (dAnsw):** The corresponding answer to a routing decision request from the Routing Peer to the Forwarding Peer. The routing decision includes the address(es) of the next hop(s) of the respective data entry and possibly additional information which is then piggybacked on the data entry.
- **delete lock entry (delLo):** Deletes a lock that is used to prevent an information request from being processed.
- **external destination entry (eDest)** Contextual information regarding external destinations held by Node Peers and not known until the time of a Node Peers initialization. For example, entries of this type contain the address of the Node Peers managing I/O Peer or the address of the benchmarks Statistics Peer.
- **external mapping entry (eMap)** Used by the I/O Peer to correctly relay entries sent by their managed Node Peers. Additionally, the mapping contains transmission interferences to apply (e.g. transmission delay) when sending entries over specific network links.
- **initialization entry (init):** Initializes a Node Peer. This could mean to trigger initial communication with other Node Peers in the network, building the initial state of the routing information base or to trigger an initial update of the routing information base (if that fits the routing algorithm). Furthermore, this type of entry is used by the Control Peer to start the benchmarks initialization process.
- **information entry (info):** Used to inform the I/O Peer that it has to inform its managed Node Peers that the benchmark has been stopped.
- **information request entry (iReq):** Is sent by the Routing Decision Peer to the Routing Information Peer in order to request information, contained in the routing information base and necessary to make a correct routing decision.
- **information request answer entry (iAnsw):** Is the corresponding answer to a routing information request from the Routing Information Peer to the Routing Decision Peer. The answer to the request includes all necessary information for the Routing Decision Peer to make a correct routing decision.
- **internal destination entry (iDest)** Holds contextual internal information of a Node Peer known at the time of the peers creation. For example, it enables sub peers to send entries to other sub peers of a Node Peer by storing their addresses i.e. it serves as internal address book for local addresses of a Node Peers sub peers.
- **internal mapping entry (iMap)** Maps addresses of Node Peers to the address of their managing I/O Peer. The I/O Peer uses this type of entry to register

at the Control Peer and to relay received entries to the correct managed Node Peer. Furthermore, it is used by the Control Peer to build, initialize and start the benchmark.

- **lock entry (lock):** Locks are a tool for the framework to keep an information request entry from being processed temporarily.
- **registration entry (reg):** Used for the registration of Node Peers and their managing I/O Peers at the Control Peer.
- **routing communication entry (rtCom):** Is sent between Node Peers in order to exchange routing information. In order to enable the framework to calculate the routing overhead, a time trace of the path from source to destination is saved in a rtCom entries co-data.
- **routing information base entry (base):** The routing information base is the knowledge base of a Node Peer. It contains all information needed to make routing decisions (e.g. a routing table or the list of neighbors of the Node Peer). It is important to note, that there exists exactly one entry of this type in an instance of a Node Peer.
- **routing communication wrapper entry (outR):** Wraps a routing communication entry for sending between Peer Spaces.
- **send entry (send):** Is used internally in the Routing Information Peer to initiate the (periodical) sending of routing communication entries to other Node Peers.
- **start entry (start):** Starts the benchmark after the defined initialization time.
- **started entry (started):** Token which enables a Node Peer to process data entries. It is generated in a Node Peer after a start entry is received.
- **statistics entry (stats):** Contains calculated statistics of the benchmark and necessary information to calculate them. This entry exists exactly once for a benchmark and is placed in the Statistics Peer at the time of its creation.
- **statistics output entry (statOut):** Triggers the output of the benchmarks calculated statistics.
- **statistics wrapper entry (outS):** Wraps extracted time traces of data entries and routing communication entries for transmission to the Statistics Peer.
- **stop entry (stop):** Used to stop the benchmark after its defined run and initialization time.
- **termination conditions entry (tCond):** Holds information that enables a Node Peer to terminate received routing information entries before they are processed.

- **timer entry (timer):** Entry that ensures that a wiring is only triggered after a configurable time interval has passed.

- **update entry (updt):** Triggers an update of the routing information base.

4.4.3 Node Peer

The Node Peer embodies a servlet in a P2P network and encapsulates the routing functionality of it. An instance of a Node Peer contains two sub peers: the Forwarding Peer (forwards incoming data entries to neighbors according to the used routing algorithm) and the Routing Peer (responsible to gather routing related data and to make routing decisions). The Forwarding Peer and the Routing Peer handle the complex tasks whereas the Node Peer itself is only responsible for rather simple ones. Tasks carried out directly by the Node Peer are: to dispatch incoming entries (data entries, routing communication entries or initialization entries) to the correct sub peer, to initialize data entries if a Node Peer is the source, to manipulate entries when they are received (e.g. increase hop count of a data entry) and to start and stop the processing of data entries (when the benchmark is started / stopped). Furthermore, the Node Peer registers incoming data entries by storing timestamps in their coordination data. This is used by the Statistics Peer for the calculation of metrics at a later stage of the benchmark.
The Node Peer contains five wirings, shown in figure 4.9, which are described in detail in the following.

4.4.3.1 W1: Data Dispatch Wiring

This wiring simply dispatches up to a configurable amount of $x \geqslant 1$ data entries to the Node Peers Forwarding Peer after the benchmark has been started.

Guards:

1. **started**: Node Peers must not process data entries when the benchmark has not been started. This is ensured by this guard.

2. **iDest**: Contains the address of the Node Peer instance. This is used to determine if the Node Peer is the source of a data entry (to determine if the data entry might be subject to initialization).

3. **data (take)**: Data entries to be dispatched by the wiring.

Actions:

1. **data**: The data entries dispatched to the Node Peers Forwarding Peer.

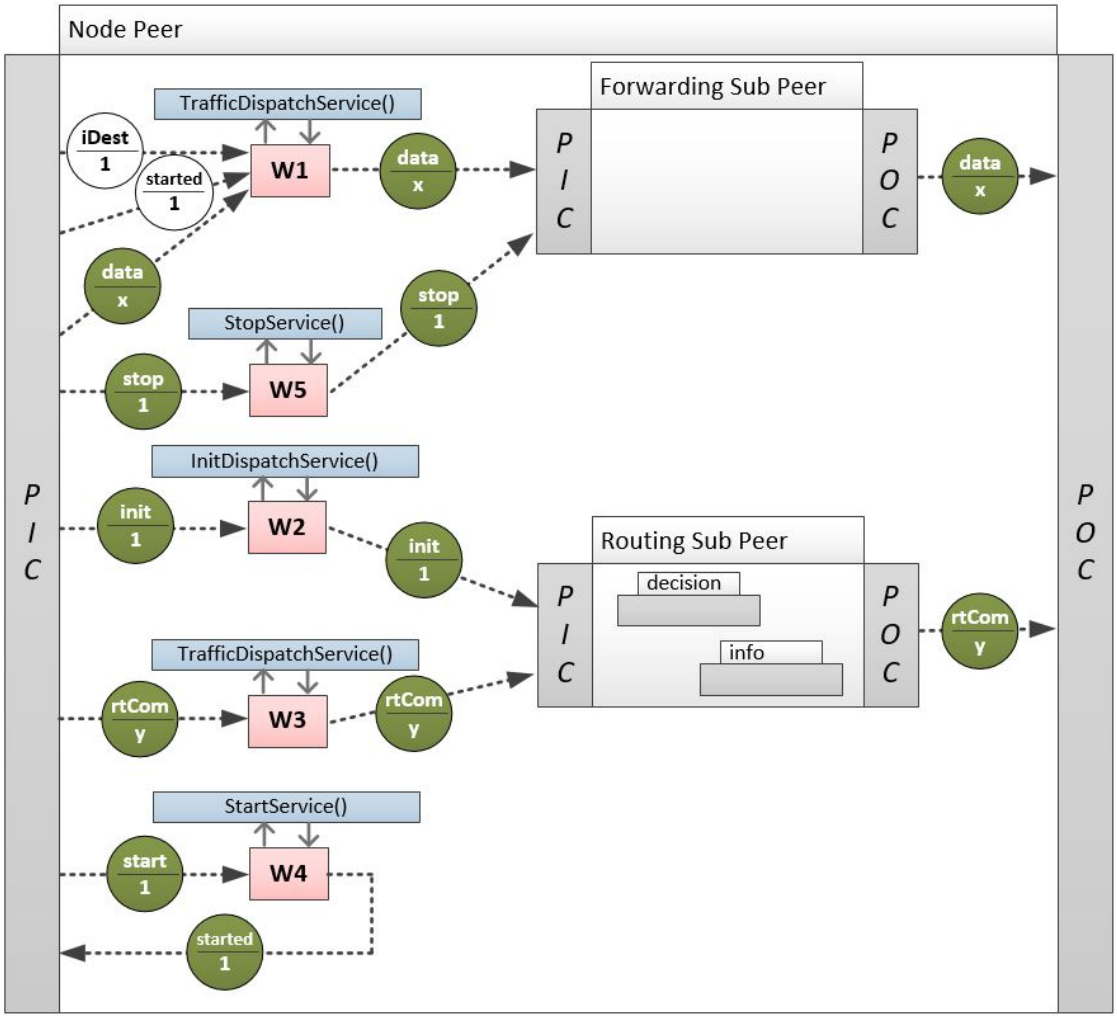

Figure 4.9: Wirings of the Node Peer

4.4.3.2 W2: Routing Dispatch Wiring

This wiring simply dispatches up to a configurable amount of $y \geqslant 1$ routing communication entries to the Node Peers Routing Peer. Note that the wiring doesn't dispatch routing communication entries to the Routing Peer itself, but directly to one of the Routing Peers sub peers, the Routing Information Peer.

Guards:

1. **rtCom (take)**: Routing communication entry to be dispatch by the wiring.

Actions:

1. **rtCom**: Routing communication entries dispatched to the Node Peers Routing Information Peer.

4.4.3.3 W3: Init Dispatch Wiring

This wiring simply dispatches an initialization entry to the Node Peers Routing Peer at initialization time.

Guards:

1. **init (take)**: The initialization entry to be dispatched by the wiring.

Actions:

1. **init**: The initialization entry dispatched to the Node Peers Routing Peer.

4.4.3.4 W4: Start Wiring

This wiring starts the benchmark by placing an entry of type "started" in the PIC of the Node Peer. This enables the Node Peer to dispatch data entries.

Guards:

1. **start (take)**: Starts the benchmark.

Actions:

1. **started**: Placed in the PIC of the Node Peer to enable the dispatching of data entries.

4.4.3.5 W5: Stop Wiring

This wiring dispatches a stop entry to the Node Peers Forwarding Peer after the run time of the benchmark has passed. The Forwarding Peer then stops requesting decisions for held data entries and sends them to the Statistics Peer for the calculation of metrics.

Guards:

1. **stop (take)**: The stop entry to be dispatched to the Node Peers Forwarding Peer.

Actions:

1. **stop**: The dispatched stop entry.

4.4.4 Forwarding Peer

The Forwarding Peer is responsible for handling incoming data entries. If a new data entry is placed in its PIC, the Forwarding Peer creates an appropriate decision request for the Routing Peer. The data entry is placed in the coordination data of the request entry. Thus, the framework is able to map the request to the corresponding data entry. Depending on the routing decision made by the Routing Peer, the Forwarding Peer either forwards the data entry to (an)other Node Peer(s) or sends it to the Statistics Peer (if the data entry has reached its final destination or the routing decision is to drop the data entry). The Forwarding Peer is able to process multiple data entries at the same time. The amount is configurable and equal for the wiring for requesting decisions and forwarding data entries.
The Forwarding Peer contains three wirings, shown in figure 4.10, which are described in detail in the following.

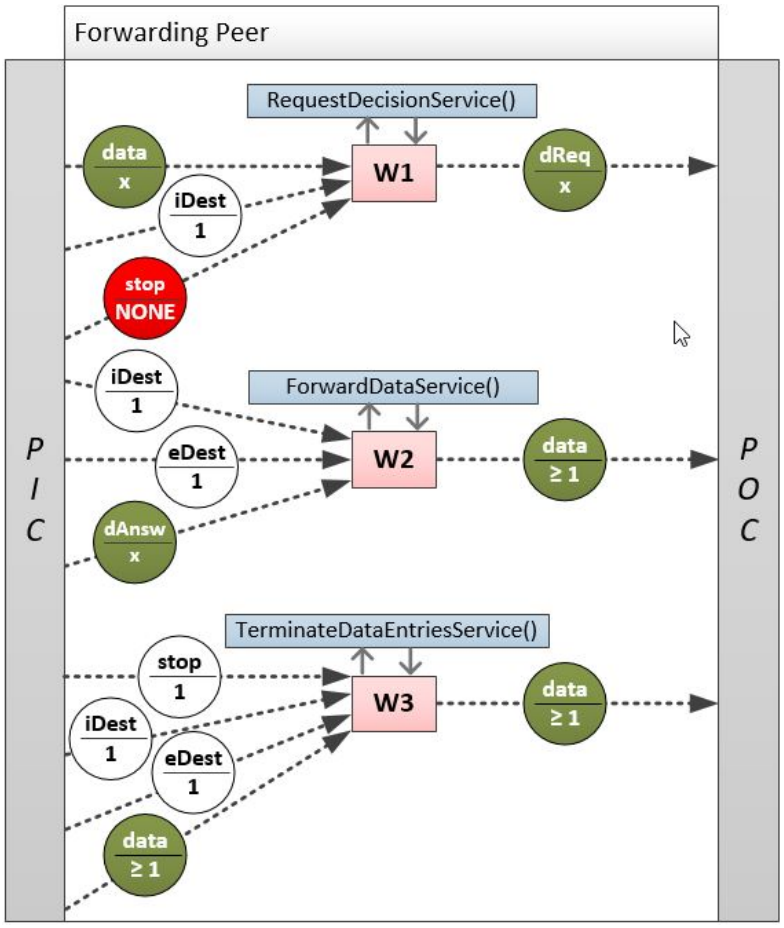

Figure 4.10: Wirings of the Forwarding Peer

4.4.4.1 W1: Request Decision Wiring

This is the wiring where the creation of routing decision requests is done. Therefore, it takes up to a configurable amount of $x \geqslant 1$ data entries from the PIC. The requests might be primitive with no additional contextual information or custom (additional information added) depending on the used routing algorithm. Furthermore, when a decision request for a data entry is created, the data entry is stored in the coordination data of the decision request. This enables the framework to determine the corresponding data entry for a decision request. The decision requests are directly placed in the PIC of the Node Peers Routing Information Peer.

Guards:

1. **iDest (read)**: This entry contains the address of the Node Peers Routing Peer. This enables the Forwarding Peer to send the decision request to the Node Peers Routing Peer.
2. **data (take)**: Data entries for which decision requests are created.
3. **stop (none)**: In order for decision requests to be created, the benchmark must not be stopped yet.

Actions:

1. **dReq**: Created decision request entries which are sent to the Node Peers Routing Peer. The decision request contains the corresponding data entry in its coordination data.

4.4.4.2 W2: Forward Data Wiring

The Forward Data Wiring handles data entries according to the routing decisions received from the Routing Peer. Up to a configurable amount of $x \geqslant 1$ data entries is processed by the wiring at a time. The corresponding data entry of a routing decision entry can be determined easily by the framework, since the data entry is contained in it. If the decision is to forward the data entry to other Node Peers, the data entry is sent to the recipients specified in the decision request. Otherwise, the data entry is sent to the Statistics Peer. The latter case occurs when the data entry has either reached its final destination with the current Node Peer or when the Routing Peer determines that the data entry should be dropped. Furthermore, the wiring optionally attaches (depending on the routing algorithm) contextual information to each data entry (e.g. the last hop address of the data entry) before forwarding it.
It is important to note, that the Forwarding Peer technically does not send entries to other Node Peers directly but via their corresponding I/O Peer.

Guards:

1. **iDest (read)**: This entry contains the address of the Node Peer itself. This enables to uses it as contextual information when optionally manipulating the data entry before sending (e.g. set the last hop address of the data entry).

2. **eDest (read)**: The entry contains the address of the Node Peers managing I/O Peer. The I/O Peer is responsible for relaying forwarded data entries and apply transmission interference.

3. **dAnsw (take)**: Answers to decision requests. The answer contains the addresses of the recipients to which the data entry should be sent. If the data entry has reached its final destination or is dropped, the decision answer entry contains solely the address of the Statistics Peer in the list of recipients. Furthermore, each answer entry contains its corresponding data entry which is subject to forward.

Actions:

1. **data**: Forwarded data entries. The data entry may be forwarded to multiple recipients (depending on the routing algorithm).

4.4.4.3 W3: Terminate Data Entries Wiring

When the defined run time of a benchmark has passed, the Node Peer must not process any of the remaining data entries in the Forwarding Peers PIC. These data entries have to be terminated. However, they are still highly relevant for the calculation of metrics. Therefore, contextual information of the remaining data entries are sent to the Statistics Peer by this wiring (via the I/O Peer).

Guards:

1. **stop (read)**: Data entries are only terminated when the benchmark run time has passed.

2. **eDest (read)**: The entry contains the address of the Node Peers managing I/O Peer. The I/O Peer is responsible for relaying forwarded data entries and apply transmission interference.

3. **iDest (read)**: This entry contains the address of the Node Peer itself. This information is used to inform the managing I/O Peer from which Node Peer the data entry was sent.

4. **data (take)**: Data entries to be terminated.

Actions:

1. **data**: Terminated data entries sent to the I/O Peer where the contextual information (traces) are extracted and sent to the Statistics Peer.

4.4.5 Routing Peer

The Routing Peer has the task to initialize itself and its sub peers. This could mean to trigger initial routing related communication with other Node Peers in the network, building the initial state of the routing information base or to trigger an initial update of the routing information base (if that fits the routing algorithm). If a routing communication entry has reached its final destination at this Node Peer instance, it is sent to the managing I/O Peer, where contextual information is extracted and sent to the Statistics Peer for the calculation of metrics at a later stage of the benchmark, after processing. Incoming routing decision requests and routing communication entries are not dispatched by the Routing Peer, but directly placed in the target sub peer by the sender.
The Routing Peer only contains a single wiring, shown in figure 4.11, which is described in detail in the following.

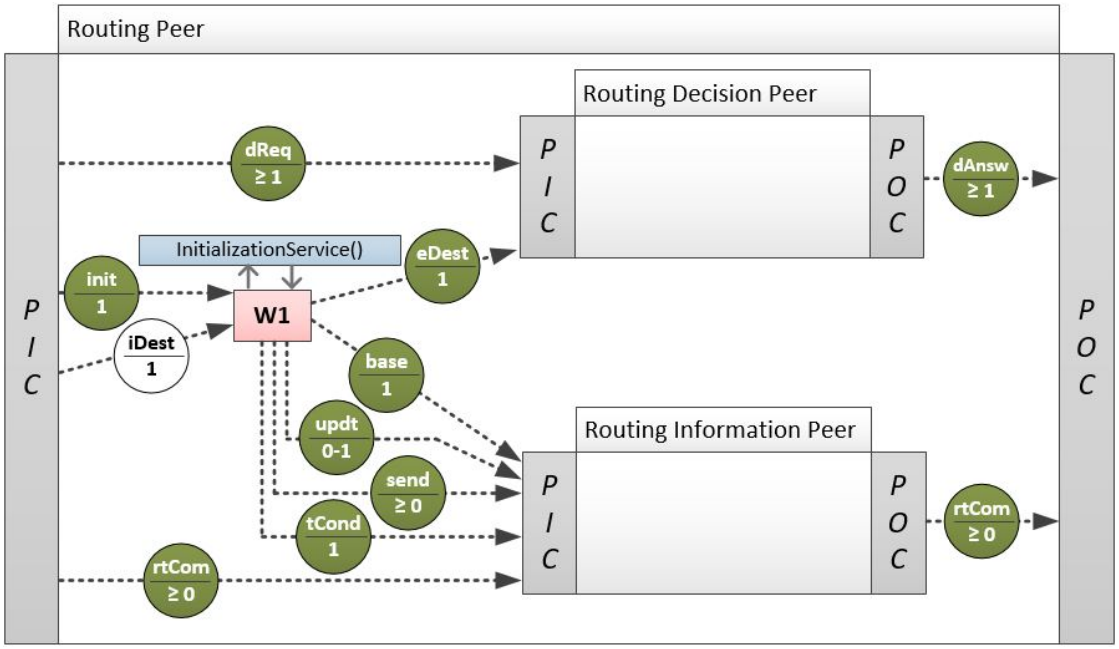

Figure 4.11: Wirings of the Routing Peer

4.4.5.1 W1: Initialization Wiring

The Routing Peer and its sub peers are initialized by this wiring. The main task of the wiring is to build the routing information base that holds all important information for the routing algorithm in order to make adequate routing decisions (e.g. list of the Node Peers neighbors in the network), to initialize the routing information base and to place it in the PIC of the Routing Information Peer. Moreover, it optionally provides the possibility to trigger initial sending of routing communication entries to other Node Peers and to immediately update the routing information base after it was dispatched to

the Routing Peers Routing Information Peer (if that fits the routing algorithm). Besides that, it sends contextual information (e.g. the address of the Statistics Peer) which is not known until the time of initialization to the Routing Decision Peer. This is not needed for the Routing Information Peer since this information is already included in the initialized routing information base.

Guards:

1. **init (take)**: The initialization entry holds all the information needed for initializing the Routing Peer and its sub peers.

2. **iDest (read)**: Holds the address of the current Node Peer which is then stored in the routing information base in the initialization process.

Actions:

1. **base**: The initialized routing information base. It serves as the knowledge base of the Node Peer. The routing information base entry always contains at least the list of neighbors of the Node Peer, the address of the Node Peer itself, the address of the frameworks Statistics Peer and the address of the corresponding I/O peer in the entries app-data.

2. **tCond**: An additional container for knowledge which is solely used to terminate routing information entries before they are processed. This also ensures that the routing information base is only taken (and therefore locked) when it is actually updated.

3. **send**: Optional entries which trigger initial sending of routing communication entries to other Node Peers after the benchmark has started.

4. **updt**: Optional entry to manipulate the routing information base right after it is dispatched to the Routing Peers Routing Information Peer.

5. **eDest**: Contains contextual information known at initialization time. More specifically, it contains the address of the corresponding I/O Peer and the Statistics Peer. This entry is sent to the Routing Decision Peer.

4.4.6 Routing Decision Peer

This Peer is the part of the pattern where the actual routing decisions are made. This includes to decide if the data entry has reached its final destination, if a termination condition is met (which would lead to the Node Peer dropping the data entry instead of forwarding it further) and to determine the recipients to which the data entry should be forwarded if that is the result of the decision process. When a routing decision for a data entry is requested by the Forwarding Peer, the Routing Decision Peer request all information needed for making the decision (e.g. the routing table or the list of the Node

Peers neighbors) by sending an information request to the Routing Information Peer. Based on the information received, the routing decision is made. The Routing Decision Peer then sends the answer, containing the result of that decision to the Forwarding Peer. The framework is able to carry out the tasks described above for up to a configurable amount of $x \geqslant 1$ decision requests / information answer entries at a single execution of the corresponding service. This limit x is equal to the limit x for handling data entries and decision answer entries in the Forwarding Peer.
The Routing Decision Peer contains two wirings, shown in figure 4.12, which are described in detail in the following.

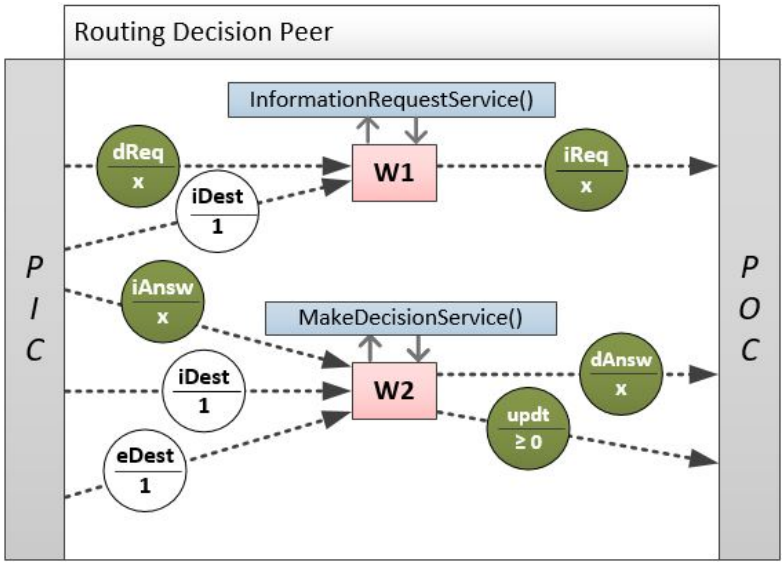

Figure 4.12: Wirings of the Routing Decision Peer

4.4.6.1 W1: Information Request Wiring

This is the wiring where the creation of information requests happens i.e. the Routing Decision Peer requests the information needed from the Routing Information Peer in order to make an adequate routing decision. This could be either primitive requests with no contextual information for the Routing Information Peer or custom requests based on the used routing algorithm and the context provided by the decision request which triggered this wiring. The information request entry and the decision answer entry contain the decision request in their coordination data, whereas the information answer entry contains the information request entry in its coordination data. Note that the decision request contains the data entry in its co-data. This nesting of entries enables the framework to map the corresponding requests and answers.

Guards:

1. **iDest (read)**: This entry contains the address of the Routing Peers Routing Information Peer. This enables the Routing Decision Peer to send the information request to the correct peer.

2. **dReq (take)**: The decision request entries for which the information request is created. Holds the corresponding data entry in its coordination data.

Actions:

1. **iReq**: Information request entries sent to the Routing Peers Routing Information Peer. Each holds the corresponding decision request in its coordination data.

4.4.6.2 W2: Make Decision Wiring

This wiring can be considered the core part of the proposed frameworks pattern. Based on the received information from the Routing Information Peer and the contextual information provided in the decision request, routing decisions are made and sent to the Forwarding Peer in the form of routing decision answer entries. The routing decision might lead to the forwarding of a data entry, for which the decision request was created, to other Node Peers or its termination. The latter is the case if the data entry has reached its final destination or a termination condition (specified by the used routing algorithm) is met (e.g. the data entry has reached a specified limit of maximum hops). When the routing decision is to drop the data entry, the routing decision answer only contains the frameworks Statistics Peer, where metrics are calculated at a later stage of the benchmark, in the list of recipients. Furthermore, the routing decision could also influence the routing information base held by the Routing Information Peer. Therefore, the wiring is able to optionally create update entries and send it to the Routing Information Peer.

Guards:

1. **iDest (read)**: This entry contains the address of the Node Peers Forwarding Peer. This enables the Routing Decision Peer to send the answer to the decision request to the correct Forwarding Peer.

2. **eDest (read)**: This entry contains the address of the frameworks Statistics Peer. When the decision is to drop a data entry, the only recipient for the next hop is this address.

3. **iAnsw (take)**: Answers to information requests. Hold the information needed to make adequate routing decisions. An information answer entry always includes the address of the Node Peer itself (of which the Routing Peer and therefore the Routing Decision Peer is a sub peer) to enable this wiring to check if the data entry (for which the decision request was created) has reached its final destination. Furthermore, it holds the corresponding information request in its coordination data.

Actions:

1. **dAnsw**: Routing decisions sent to the Forwarding Peer. Each answer entry contains the corresponding decision request in its coordination data.

2. **updt**: Entries optionally created and sent to the Routing Information Peer for an asynchronous update of the routing information base, triggered by the routing decision.

4.4.7 Routing Information Peer

The knowledge base (routing information base) contains all routing relevant information needed in order for the Routing Decision Peer to make adequate routing decisions (e.g. routing table, list of the Node Peers neighbors). The Routing Information Peer is responsible for managing it. Thus it updates it, if needed, and provides a copy of it partially or fully (depending on the routing algorithm) on request to the Routing Decision Peer. Updates to the routing information base could be based on information received from other Node Peers, resulting from made routing decisions by the Routing Decision Peer or internally triggered by the Routing Information Peer itself.
Furthermore, the Routing Information Peer might not only receives update relevant data from other Node Peers in a passive manner, it is also able to actively request information if needed. However, to enable blocking operations (i.e. an information request is not processed until requested information is received) when information is requested actively, a specific option has to be activated in the frameworks configuration. If the option is activated, the Routing Information Peer partially uses different wirings. Thus, there are two versions of the Routing Information Peer. The actual version is chosen at creation time of a benchmark. More details regarding the frameworks configuration can be found in section 5.5.
When it comes to intelligent routing algorithms, the Routing Information Peer is the component which spawns, registers and interacts with intelligent agents that travel the network.
The wirings of the Routing Information Peer are shown in two parts in figures 4.13 and 4.14 and described in detail in the following. Note that figure 4.13 shows two different versions of the wiring to answer information requests. The wiring which is actually used depends, if blocking operations are needed in the implemented routing algorithm or not.

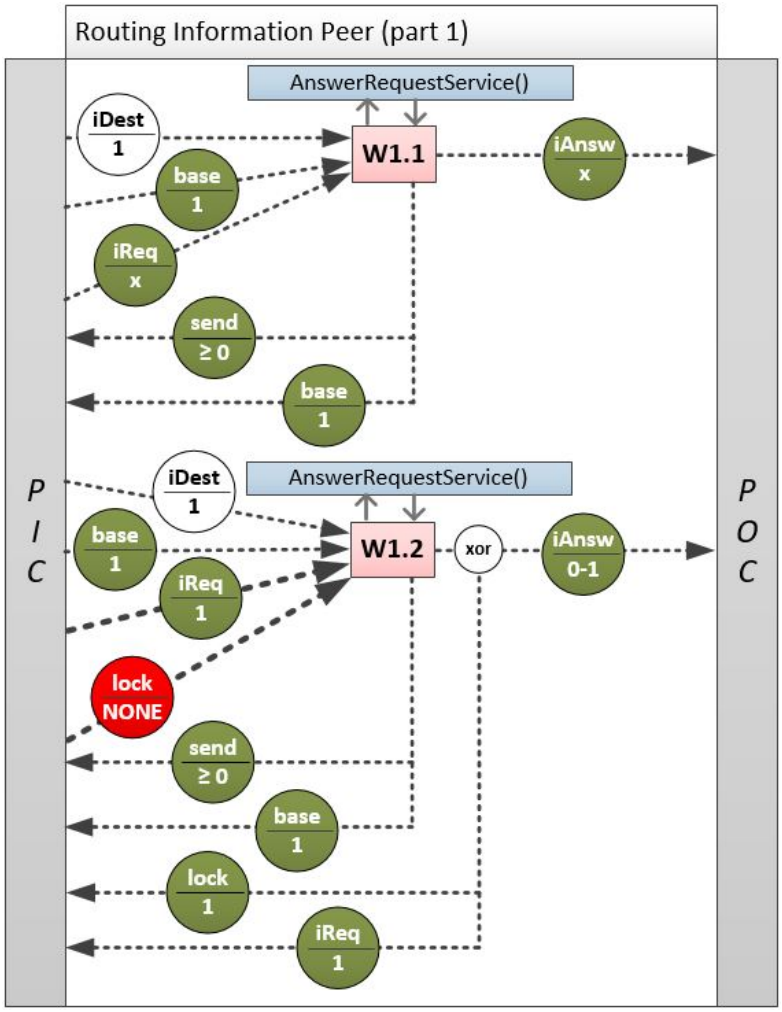

Figure 4.13: Both version of the Routing Information Peers wiring to answer information requests.

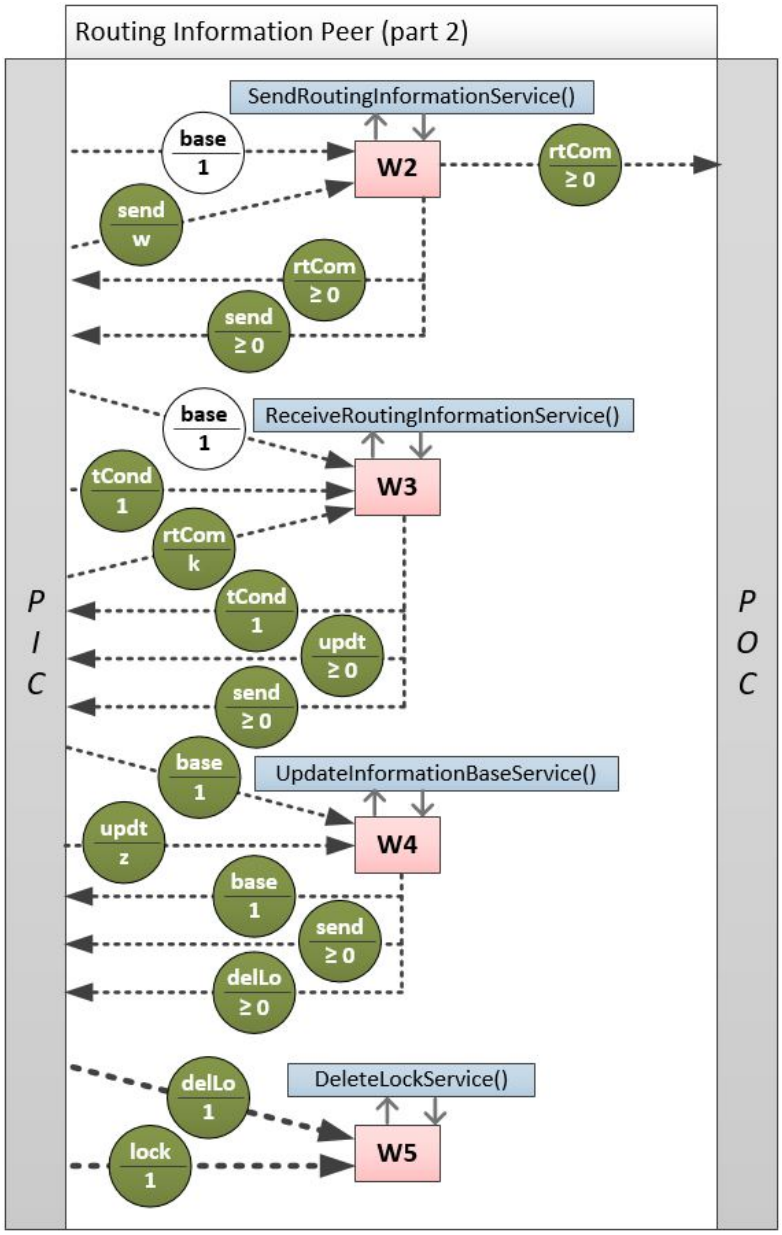

Figure 4.14: Wirings of the Routing Information Peer (part 2)

4.4.7.1 W1.1: Information Answer Wiring (version 1)

This is the standard wiring used by the Routing Information Peer for processing information requests received from the Routing Decision Peer. Although it does not allow blocking operations (blocking the processing of an information request until actively requested information for updating the routing information base has been received), it

has the major advantage that multiple requests can be processed with one execution of the wirings service. Thus, it prevents this wiring from being the performance bottleneck for processing data entries at a Node Peer.
The routing information base as well as up to a configurable amount of $x \geqslant 1$ information requests are taken from the PIC of the Routing Information Peer. Each information request is answered according to the used routing algorithm. The answers are placed as routing information answer entries into the PIC of the Routing Decision Peer.
Furthermore, the wiring is able to update the routing information base. This is needed for algorithms which synchronously update the routing information base based on information, provided by data entries context, contained in information requests. Additionally, the wiring is able to optionally create entries of type "send" to initiate communication with other Node Peers based on the received information request.

Example 4.4.4 *Assume the implemented routing algorithm is a flooding algorithm where each originated data entry has an unique ID and each of theses data entries is only allowed to be forwarded once by a Node Peer. Therefore, a list of IDs of received data entries has to be kept in the routing information base. In order to avoid forwarding a data entry more than once, the list has to be updated synchronously after a information request has been processed.*

Guards:

1. **iDest (read)**: This entry contains the address of the Routing Peers Routing Decision Peer. This enables the Routing Information Peer to send the answer to the information request to the correct Routing Decision Peer.

2. **base (take)**: This entry contains the full knowledge base of the Node Peer. It is used by the wiring to answer the received information request.

3. **iReq (take)**: Information requests to be answered by this wiring.

Actions:

1. **iAnsw**: Answer to the information requests. Each answer entry always contains at least the address of the Node Peer (of which the Routing Peer and therefore the Routing Information Peer is a sub peer) to enable the Routing Decision Peer to determine if the data entry, for which the request chain was initiated in the first place, has reached its final destination at the current Node Peer.

2. **send**: Optionally, these entries are built to initiate communication with other Node Peers (e.g. to request additional information from other network nodes).

3. **base**: The routing information base is placed back in the PIC after a possible update.

4.4.7.2 W1.2: Information Answer Wiring (version 2)

If blocking operations are needed when it comes to answering routing information requests, this version of the wiring is needed. Therefore, a specific option has to be activated in the configuration of the framework. Although it provides more utility than the standard alternative (wiring W1.1), its usage comes with a major drawback. Information request entries can only be processed one by one. Due to that fact, the wiring becomes the frameworks bottleneck performance-wise.
This version of the Information Answer Wiring reads the routing information base from the PIC of the Routing Information Peer and checks if it is able to process the information request. If that is the case, the requested information is sent in the form of a routing information answer entry to the Routing Decision Peer.
Otherwise, the processing of the request is put to hold and a lock entry is created. The lock prevents the information request from being processed until the lock is removed. Example reasons for this case are when the information needed in order to answer the information request has to be specifically requested from another Node Peer or if the wiring should wait for a general update of the routing information base (without a specific request).
The wiring is also able to update the routing information base. Furthermore, independently of the wiring being able to process the request, the wiring is able to optionally create entries of type "send" to initiate communication with other Node Peers (e.g. to request the information missing in the routing information base for the wiring to be able to answer the request satisfactorily). This can be useful in multiple scenarios (e.g. when a lock should be created to wait for an update of the routing information base which is not triggered by a specific request to other Node Peers). An information request is only processed if there exists no corresponding lock entry (i.e. the information request has not been processed yet or the processing is not put on hold) in the PIC of the Routing Information Peer.
Routing information request entries and lock entries are matched via their flow-ID by the framework. Thus the guards for these two entry types are flow-dependent.

Guards:

1. **iDest (read)**: This entry contains the address of the Routing Peers Routing Decision Peer. This enables the Routing Information Peer to send the answer to the information request to the correct Routing Decision Peer.
2. **base (take)**: This entry contains the full knowledge base of the Node Peer. It is used partially or fully by the wiring to answer the received information request. Furthermore, it might be updated.
3. **iReq (take)**: The information request to be answered by this wiring.
4. **lock (none)**: This entry ensures that if the processing of a information request is put on hold, it is not processed again until the information base contains the information needed to answer the request sufficiently.

Actions:

1. **iAnsw**: The answer entry which is created if the wiring is able to process the information request. The entry always contains at least the address of the Node Peer (of which the Routing Peer and therefore the Routing Information Peer is a sub peer) to enable the Routing Decision Peer to determine if the data entry, for which the request chain was initiated in the first place, has reached its final destination at the current Node Peer.

2. **iReq**: The information request is put back in the PIC of the Routing Information Peer if it currently can't be processed.

3. **lock**: If the wiring is not able to process the information request currently, a lock entry to temporarily block the processing of the request is created and placed in the PIC of the Routing Information Peer.

4. **send**: Optionally, these entries are built to initiate communication with other Node Peers (e.g. to request additional information). It is important to note that the creation is independent of the wiring being able to answer the request or not.

5. **base**: The routing information base is placed back in the PIC after a possible update.

4.4.7.3 W2: Send Routing Information Wiring

The wiring takes up to a configurable amount of $w \geqslant 0$ "send" entries and executes its service. This might lead to the creation of routing communication entries which are sent to other Node Peers for inter-node-peer communication purposes (e.g. to inform another P2P nodes about an update of the information base or to send an update request).
Furthermore, the wiring optionally creates entries of type "send" to enable periodical sending of routing information.

Example 4.4.5 *Assume the implemented routing algorithm is an intelligent one that uses agents which travel the network and gather routing relevant information. The Send Information Wiring is used to forward an agent to its next hop. Furthermore, entries of type "send" are placed in the PIC of the Routing Information Peer in order to trigger periodical spawning of the intelligent agents.*

Guards:

1. **base (read)**: This entry contains the full knowledge base of the Node Peer. It contains the list of neighbors of the Node Peer and is therefore needed by the wiring. The wiring may also uses additional information of the routing information base. Additionally, the entry holds the address of the corresponding I/O peer that relays the sent entries to the correct target.

2. **send (take)**: Triggers the sending of routing relevant information or agents to other Node Peers.

Actions:

1. **rtCom**: Entries sent to other Node Peers (via the managing I/O Peer). The information contained could trigger an update of the other Node Peers information base or simply an answer to the request sent. Entries of this type can also be placed in the PIC of the Routing Information Peer in order to update the own routing information base via the Receive Routing Information Wiring.

2. **send**: These entries are built optionally to initiate communication and collaboration with other Node Peers periodically. Therefore, they are placed in the PIC of the Routing Information Peer (most likely with a TTS set).

4.4.7.4 W3: Receive Routing Information Wiring

This wiring represents the receiving end of communication between Node Peers and therefore handles up to $k \geqslant 0$ incoming routing communication entries per execution. Furthermore, it decouples the receiving of routing information from the process of updating the information base. This is especially advantageous for the performance of the framework, since not every received routing information entry leads to an update of the information base, but each execution of the Update Information Base Wiring locks the routing information base, which is needed for answering routing information requests. To enable the wiring to filter routing information entries that do not lead to an update of the routing information base, it possesses its own private knowledge base in the form of a termination condition entry. Terminated routing information entries are sent to the frameworks Statistics Peer (via the Node Peers managing I/O Peer), where metrics are calculated at a later stage of the benchmark. If the routing information base has to be updated based on the received information, an update request that includes the update information is created and placed in the PIC of the Routing Information Peer. Additionally, the wiring is able to initiate communication with other Node Peers by creating entries of type "send".

Example 4.4.6 *Assume an intelligent routing algorithm that uses agents sent by nodes to inform other nodes about the delay between them on possible paths (experienced by the agents). Furthermore, assume that the algorithm only allows agents to travel a maximum amount of $p \geqslant 1$ hops. If a registered agent has reached the defined limit, it is ensured that it does not trigger an update of the routing information base, but gets terminated.*

Example 4.4.7 *Assume an intelligent routing algorithm that uses agents that experience and report paths in the network to P2P nodes. In order to do that, an agent has the possibility to manipulate the routing table contained in the routing information base of a Node Peer. If the agent decides to manipulate the routing information base, it is placed in the form of an update request (entry of type "updt") in the PIC of the Routing Information*

Peer. Otherwise, it might continues to travel the network by being placed as "send" entry in the PIC.

Guards:

1. **base (read)**: This entry contains the full knowledge base of the Node Peer. However it is only read and can be used as additional information source to decide what to do with a received routing information entry.
2. **tCond (take)**: Additional knowledge base used by this wiring to store information used to decide if a received routing information entry should be terminated.
3. **rtCom (take)**: Received entries that contain routing relevant information. Might lead to updates of the routing information base.

Actions:

1. **update**: Optionally created to update the routing information base held by the Node Peers Routing Information Wiring.
2. **send**: These entries are built optionally to initiate communication with other Node Peers.
3. **tCond**: The termination conditions are placed back in the PIC of the Routing Information Peer after a possible update.

4.4.7.5 W4: Update Information Base Wiring

This wirings major task is to update the routing information base based on received update requests. To do this, the wiring takes the routing information base from the PIC of the Routing Information Peer and therefore locks it from all other access. Thus, no routing information requests can be processed by the Routing Information Peer until the update is done.
If the option for blocking routing information requests is enabled, the wiring is able to remove locks after the routing information base has been updated.

Example 4.4.8 *Assume an intelligent routing algorithm that uses agents sent by nodes to inform other nodes about the delay between them on possible paths (experienced by the agents). When an agent reaches a network node, it trigger updates of the routing information base. Therefore, after being registered by the Receive Routing Information Wiring, they are processed as entries of type "updt" by the Update Information Base Wiring. After the update, the agents move on with their travel in the network by being placed as entries of type "send" in the PIC of the Routing Information Peer (and then forwarded by the Send Routing Information Wiring).*

Guards:

1. **base (take)**: The routing information before the applied updates.

2. **updt (take)**: These entries trigger an update of the routing information base.

Actions:

1. **base**: The updated routing information base is returned to the PIC of the Routing Information Peer.
2. **send**: Optionally, each update of the routing information base could lead to the creation of entries of type "send" which initiate the sending of information to other Node Peers.
3. **delLo**: To delete a lock in the PIC of the Routing Information Peer, an entry of type "delLo" with the same flow-ID as the lock to remove is placed in the PIC of the Routing Information Peer. One of these entries is created per lock to delete.

4.4.7.6 W5: Delete Lock Wiring

The wirings simply deletes a specific lock when triggered. The lock blocks a corresponding information request from being processed by the Routing Information Peers Answer Request Wiring. An entry which triggers the deletion and the corresponding lock entry are mapped via their flow-ID by the framework. Therefore, both of the wirings guards are flow-dependent.

Guards:

1. **delLo (take)**: This entry triggers the deletion of a specific lock entry.
2. **lock (take)**: Lock entry to be deleted.

Actions: None

4.5 Additional Framework Components

These Peers are not directly part of the core pattern. However, they are used in in the framework as tools to create and initialize the benchmark, offer a realistic environment (e.g. by simulating transmission interference), for decoupling communication concerns from the core pattern and to provide flexibility to the framework.

4.5.1 I/O Peer

The I/O Peer decouples the inter-peer communication between Node Peers in the framework from the core pattern. Each Node Peer has an associated I/O peer which is responsible for relaying the sent and received data and routing communication entries. There exists exactly one I/O Peer per Node Peer Space that manages all Node Peers

contained in it.
It therefore holds two mapping entries in its PIC: the internal mapping (contained in a single entry of type “iMap”) and the external mapping (contained in a single entry of type “eMap”). The internal mapping is used to distribute received entries correctly to managed target Node Peers and to register the I/O Peer and its managed Node Peers at the Control Peer. It maps the addresses of Node Peers to the address of the I/O Peer. Thus, the internal mapping states which Node Peers communication is handled by the I/O Peer.
In contrast, the external mapping is used for relaying data and routing communication entries sent by the I/O Peers managed Node Peers. It holds all network links between the managed Node Peers and their neighbors as well as the attributes of the links. Therefore, for each Node Peer s managed by the I/O Peer, a source and destination pair (s, d), where d are all neighbors of this Node Peer, is mapped to a target address. This target address could either be the address of the target Node Peer (if source and destination are both managed by the same I/O Peer) or the address of the I/O Peer which manages the target Node Peer. Furthermore, the external mapping is able to hold additional information for each link between two Node Peers. This can be used by the framework to simulate transmission interference (e.g. transmission delay or drop rate of entries) and therefore enables the framework to provide a more realistic environment for benchmarking.
Additionally, to all outgoing data and routing communication entries relayed by the I/O Peer, a timestamp of the sending time is added to the trace in the coordination data for the calculation of statistics at a later stage of the benchmark. The wiring also informs its managed Node Peers when a benchmark is stopped.
The I/O Peer contains seven wirings, shown in figures 4.15 and 4.16, which are described in detail in the following.

4.5.1.1 W1: Send Data Wiring

The wiring is responsible for relaying up to a configurable amount of $i \geqslant 1$ data entries, which are sent by the I/O Peers managed Node Peers, to other Node Peers. In order to do that, it determines the source and next hop destination of the communication and reads from the external mapping (contained in an entry of type “eMap”) to which peer the data entry should be sent. If source and target Node Peer are managed by the same I/O Peer, this address is the one of the target Node Peer. The data entry is then directly sent to the target Node Peer. However, if the target Node Peer is managed by another I/O Peer, this address it the one of the managing I/O Peer. In this case, the I/O Peer wraps the contents of the data entry in a entry of type “outD” and sends it to the I/O Peer that manages the target Node Peer. The entry of type “outD” allows the I/O Peer on the receiving end to distinguish between data entries which are sent or received. Furthermore, the I/O Peer reads additional information to this link between source and destination from the external mapping to be able to apply transmission interference (e.g. transmission delay). Before relaying an entry, the wiring adds a timestamp to the trace in its coordination-data.

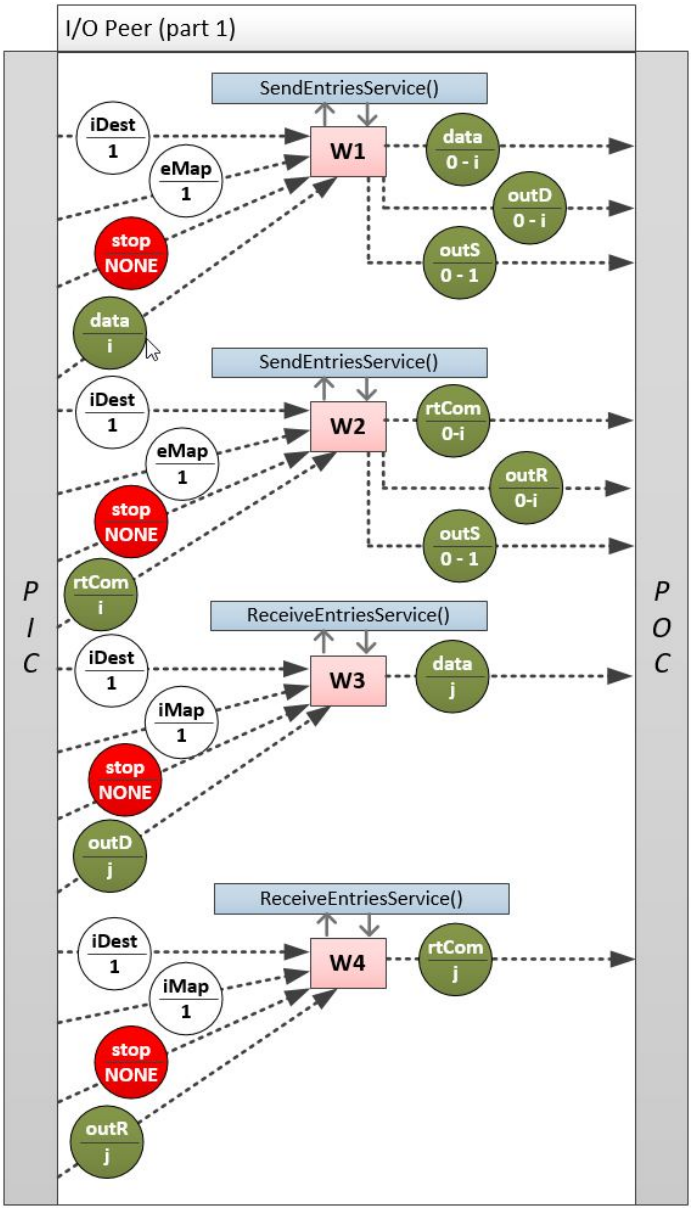

Figure 4.15: Wirings of the I/O Peer (part 1)

Guards:

1. **iDest (read)**: This entry contains the address of the I/O Peer. The wiring uses this information to determine if the target Node Peer exists in the same Node Peer Space as the I/O Peer. If that is the case, it is implicit that the target Node Peer is also managed by the same I/O Peer.

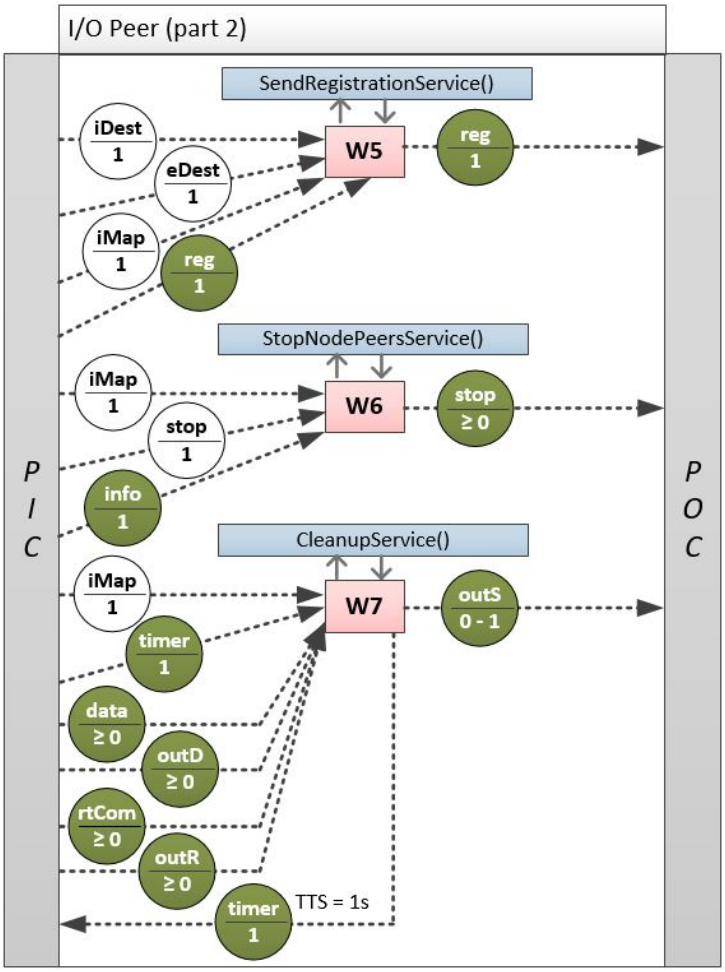

Figure 4.16: Wirings of the I/O Peer (part 2)

2. **eMap (read)**: Contains the external mapping. It is used by the wiring to determine to which peer the data entry should be sent and to check if and which transmission interference should be applied.

3. **stop (none)**: The wiring only sends data entries to other Node Peers if the benchmark has not been stopped yet.

4. **data (take)**: Data entries to be relayed by the I/O Peer.

Actions:

1. **data**: Entries sent to target Node Peers if they are managed by the this I/O Peer.

2. **outD**: Entries sent to the target Node Peers managing I/O Peer if they are managed by another I/O Peer than the source Node Peer.

4.5.1.2 W2: Send Routing Communication Wiring

This wiring does exactly the same as the Send Data Wiring, except it is responsible for relaying routing communication entries.

Guards:

1. **iDest (read)**: This entry contains the address of the I/O Peer. The wiring uses this information to determine if the target Node Peer exists in the same Node Peer Space as the I/O Peer. If that is the case, it is implicit that the target Node Peer is also managed by this I/O Peer.
2. **eMap (read)**: Contains the external mapping. It is used by the wiring to determine to which peer the routing communication entry should be sent and to check if and which transmission interference should be applied.
3. **stop (none)**: The wiring only sends routing communication entries to other Node Peers if the benchmark has not been stopped yet.
4. **rtCom (take)**: Routing communication entries to be relayed by the I/O Peer.

Actions:

1. **rtCom**: Entires sent to target Node Peers if they are managed by the this I/O Peer.
2. **outR**: Entries sent to the target Node Peers managing I/O Peer if they are managed by another I/O Peer than the source Node Peer.

4.5.1.3 W3: Receive Data Entries Wiring

When a Node Peer sends a data entry via its managing I/O Peer to a Node Peer managed by another I/O Peer, the content of the data entry is placed in form of an entry of type "outD" in the PIC of the receiving Node Peers managing I/O Peer. Due to the entries placement in the I/O Peer PIC, the Receive Data Entries Wiring is triggered. The wiring reads the internal mapping, contained in an entry of type "iMap" which is read from the I/O Peers PIC, to check if recipient Node Peer is really managed by the I/O Peer. If thats the case, it simply forwards the contents of the "outD" entry in the form of a data entry to the recipient Node Peer. The amount of simultaneously processed entries of type "outD" is configurable up to an amount of $j \geqslant 1$.

Guards:

1. **iDest (read)**: This entry contains the address of the I/O Peer. It is used for an additional check to ensure that the recipient Node Peer exists on the same Peer Space as the I/O Peer. This implies that it is managed by the I/O Peer.

2. **iMap (read)**: Mapping which contains the address of all Node Peers managed by the I/O Peer.

3. **stop (none)**: The wiring only relays data to the I/O Peers managed Node Peers if the benchmark has not been stopped yet.

4. **outD (take)**: Data entries to relay. The entry type is used to enable the I/O Peer to distinguish data entries sent by Node Peers managed by it and those ones received from Node Peers managed by other I/O Peers.

Actions:

1. **data**: Entries forwarded to the target Node Peer.

4.5.1.4 W4: Receive Routing Entries Wiring

This wiring does exactly the same as the Receive Data Entries Wiring, except it is responsible to forward routing communication entries, which are received as entries of type "outR" to the recipient Node Peer.

Guards:

1. **iDest (read)**: This entry contains the address of the I/O Peer. It is used for an additional check to ensure that the recipient Node Peer exists on the same Peer Space as the I/O Peer. This implies that it is managed by the I/O Peer.

2. **iMap (read)**: Mapping which contains the address of all Node Peers managed by the I/O Peer.

3. **stop (none)**: The wiring only relays routing communication to the I/O Peers managed Node Peers if the benchmark has not been stopped yet.

4. **outR (take)**: Routing communication entries to relay. The entry type is used to enable the I/O Peer to distinguish routing communication entries sent by Node Peers managed by it and those ones received from Node Peers managed by other I/O Peers.

Actions:

1. **rtCom**: Entries forwarded to the target Node Peer.

4.5.1.5 W5: Register Wiring

This wiring sends the I/O Peers internal mapping and its address to the Control Peer. Thus, it registers itself and the managed Node Peers at the Control Peer.

Guards:

1. **iDest (read)**: This entry contains the address of the I/O Peer itself.
2. **eDest (read)**: This entry contains the address of the Control Peer.
3. **iMap (read)**: Mapping which contains the address of all Node Peers managed by the I/O Peer.
4. **reg (take)**: This entry triggers the registration process.

Actions:

1. **reg**: Registration entry sent to the frameworks Control Peer. It contains the internal mapping as well as the address of the I/O Peer.

4.5.1.6 W6: Stop Node Peers Wiring

This wiring informs all of the I/O Peers managed Node Peers when a benchmark has been stopped.

Guards:

1. **stop (read)**: Stop entry received by the frameworks Control Peer. This entry is kept in the I/O Peers PIC in order to force the peer to terminate all received entries (and therefore sent their traces to the Statistics Peer).
2. **iMap (read)**: Mapping which contains the address of all Node Peers managed by the I/O Peer.
3. **info (take)**: Info entry received by the frameworks Control Peer. Since this entry is consumed by the wiring, it ensures that the wiring is only triggered once.

Actions:

1. **stop**: To each of the I/O Peers managed Node Peers a stop entry is sent.

4.5.1.7 W7: Cleanup Wiring

After a benchmark has been stopped, the wiring terminates all received data and routing communication related entries and sends its contextual information to the Statistics Peer. The wiring is triggered periodically after a time interval of 1 second has passed.

Guards:

1. **iMap (read)**: Mapping which contains the address of all Node Peers managed by the I/O Peer.
2. **timer (take)**: Entry that ensures that the wiring is only triggered after a time interval of 1 second has passed.
3. **data, outD, rtCom, outR (take)**: entries from which benchmark relevant contextual information is extracted before being terminated.

Actions:

1. **outS**: The gathered contextual information is wrapped in this entry and sent to the Statistics Peer for metrics calculation.

4.5.2 Statistics Peer

The Statistics Peer is used by the framework to calculate the values of the benchmarks metrics. Therefore, it uses relevant information, extracted from data entries and routing communication entries and wrapped in entries of type "outS". Furthermore, this peer also includes a wiring for the output of the calculated statistics. The calculated statistics are saved in an entry of type "stats" which is held in the PIC of the Statistics Peer. There only exists one instance of the Statistics Peer for each benchmark and exactly one "stats" entry in the PIC of this peer at all times.
The Statistics Peer contains two wirings, shown in figure 4.17, which are described in detail in the following.

4.5.2.1 W1: Calculate Statistics Wiring

The wiring calculates the benchmarks statistics based on the received contextual information. The calculation results are stored and outputted by another wiring of the Statistics Peer after the benchmark has ended and all relevant information has been included in the calculation.

Guards:

1. **outS (take)**: Entries that hold the information relevant for the calculation.
2. **stats (take)**: The overall statistics of the current benchmark before the current "outS" entry is taken into account. Note that there exists exactly one entry of this type at all times in the PIC of the Statistics Peer.

Actions:

1. **stats**: Overall statistics of the current benchmark after the calculation.

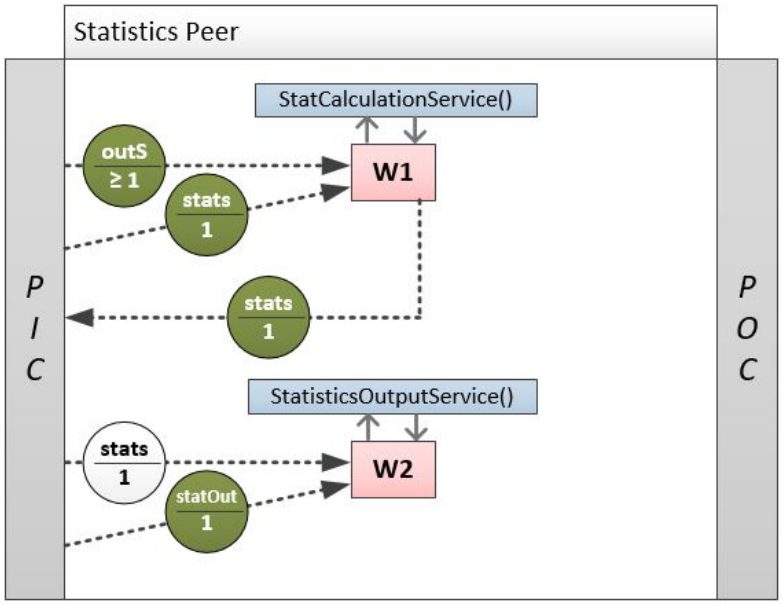

Figure 4.17: Wirings of the Statistics Peer

4.5.2.2 W2: Statistics Output Wiring

This wiring solely serves the purpose of outputting the collected statistics regarding the benchmarks metrics. Statistics are read from an entry of type "stats". In which format the output actually happens is not fixed, but defined by the wirings service.

Guards:

1. **stats (read)**: Calculated statistics of the current benchmark.
2. **statsOut (take)**: This entry triggers the output of the benchmarks statistics.

Actions: None

4.5.3 Control Peer

The Control Peer is the control center of the routing framework. Therefore, the majority of its features are configurable. It generates the topology of the network and the attributes of the network links for a benchmark. Based on the generated network, the Control Peer initializes the Node Peers, their managing I/O Peers and the Statistic Peer. Furthermore, the data entries, which are sent from a source and destination during the benchmark, are created and placed in the PIC of their determined source Node Peer. Additionally, the Control Peer starts the benchmark after a configurable initialization time a and stops it after the benchmarks run time b. 30 seconds after the benchmark has been stopped, the output of the calculated statistics is triggered by the Control Peer. This ensures that the

framework has enough time to include entries, which have been terminated due to the stopping of the benchmark, in the calculation of the statistics.
The Control Peer contains four wirings, shown in figure 4.18, which are described in detail in the following.

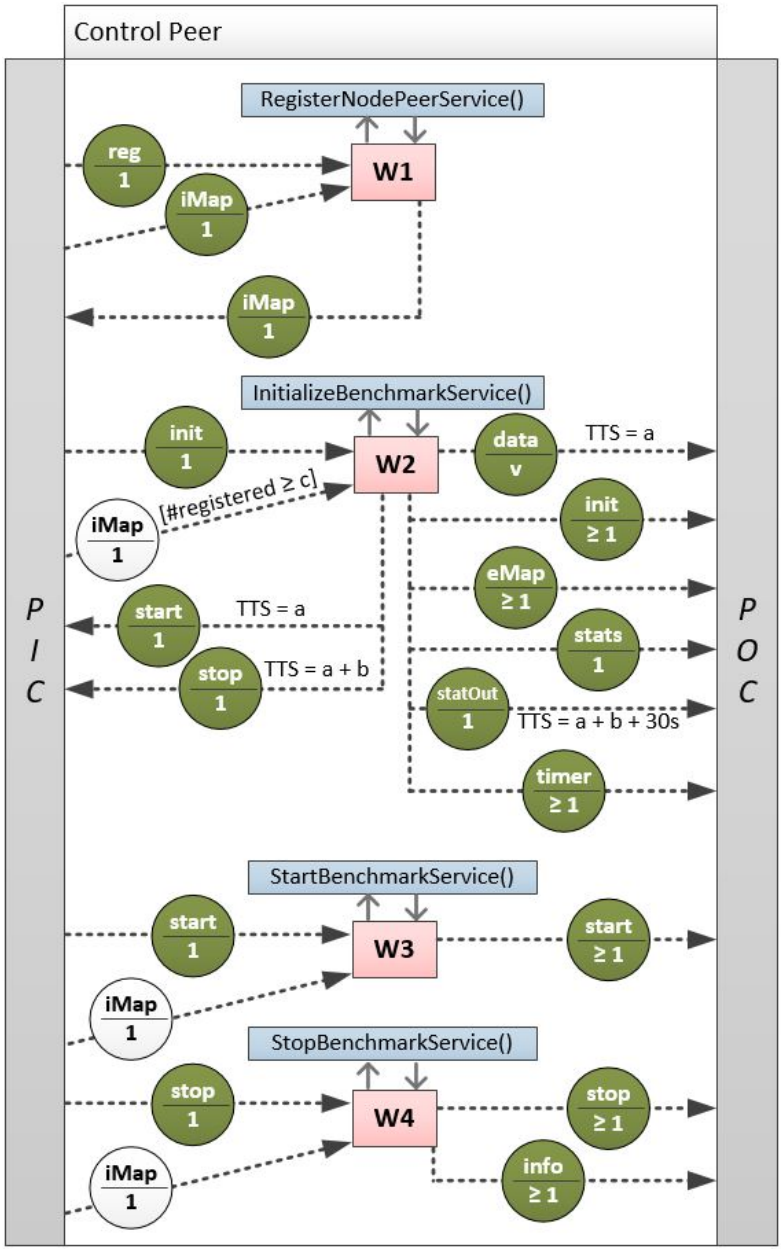

Figure 4.18: Wirings of the Control Peer

4.5.3.1 W1: Register Node Peer Wiring

I/O Peers register Node Peers at the Control Peer by informing it about their own address and those of all the Node Peers they manage. This happens by sending an I/O Peers internal mapping contained in an entry of type "reg". The wiring collects these mappings, adds them to a list containing all internal mappings received and saves them as entry of type "iMap" in the PIC of the Control Peer. Additionally, the number of registered Node Peers is tracked in this entry.

Guards:

1. **reg (take)**: Registration entry sent by an I/O Peer containing its own address and those of the managed Node Peers.
2. **iMap (take)**: Entry which contains all internal mappings received from I/O Peers before the newly received internal mapping is added.

Actions:

1. **iMap**: Entry which contains all internal mappings received from I/O Peers after the newly received internal mapping is added. Contains the number of overall registered Node Peers in its coordination-data.

4.5.3.2 W2: Initialize Benchmark Wiring

The Initialize Benchmark Wiring builds the benchmarks Peer-to-Peer network topology and initializes all involved framework components. The wiring is only triggered if at least the amount of Node Peers defined in the configuration (the configured size c of the benchmarks network) have registered at the Control Peer.
First, the network topology is generated. Not only the type of topology (e.g. ring, scale-free) is configurable, but also the number of network nodes (Node Peers). The attributes of each network link are calculated randomly in configurable interval boundaries.
Based on the created Peer-to-Peer network topology, the external mappings of all involved I/O Peers are automatically generated and placed in their PICs.
Additionally, initialization entries for all Node Peers of the benchmarks network are created and placed in their PIC. An initialization entry sent to a Node Peer includes a list of its neighbor Node Peers, the address of its managing I/O Peer and the address of the benchmarks Statistic Peer.
Furthermore, a configurable amount of data entries, which are sent from a random source to a random destination, is created and placed in the PICs of the source Node Peers.
Then the wiring initializes the statistics entry in which the calculated statistics of the benchmark are going to be stored, places it in the PIC of the Statistics Peer and creates an entry of type "statOut" which triggers the output of the measured metrics after the benchmark has ended.
Lastly, the wiring creates an entry of type "start" which serves as a trigger to start the benchmark and an entry of type "stop" to stop it, both with a configurable TTS set.

The TTS a defines how long the Node Peers have time to initialize before the actual benchmark starts (i.e. to communicate with each other to build their knowledge base), while the TTS b defines the length of the benchmarks run time (i.e. to send data entries from their source to their destination).

Guards:

1. **init (take)**: Entry which triggers the creation and initialization of the benchmark. It contains the values of the benchmarks configurable parameters and the address of the Statistics Peer.

2. **iMap (read)**: Contains all received internal mappings. The guard of this wiring is only triggered if the amount of registered Node Peers (stored in a counter in the coordination-data of this entry) equals or is greater than the amount c of expected Node Peers for the benchmark (network size defined in the configuration).

Actions:

1. **data**: Configurable amount $v \geqslant 1$ of data entries, with a random destination, placed in a random source Node Peer.

2. **init**: For each Node Peer involved in the benchmark, an initialization entry containing a list of the Node Peers neighbors, the address of its managing I/O Peer and the address of the Statistics Peer is placed in its PIC.

3. **eMap**: For each I/O Peer involved in the benchmark, the external mapping is generated and placed in the I/O Peers PIC. The mapping is used by an I/O Peer to determine to which peers (Node Peer or other I/O Peer) an data or routing communication entry should be sent and to check if and which transmission interference should be applied.

4. **stats**: Initialized entry in which the benchmarks statistics are going to be stored.

5. **statOut**: Triggers the output of the benchmarks statistics after $a + b + 30000$ milliseconds, where a is the benchmarks initialization time and b is the benchmarks run time.

6. **timer**: Entry sent to each I/O Peer to ensures that their Cleanup Wiring is only triggered once each second after the benchmark is stopped.

7. **start**: Triggers the start of the benchmark after a configurable time $a \geqslant 0\ ms$ has passed.

8. **stop**: Stops the benchmark after the initialization time a and the run time b, both configurable, have passed.

4.5.3.3 W3: Start Benchmark Wiring

After the benchmarks initialization time-frame $a \geqslant 0\ ms$ (when the TTS of the start trigger has passed), the wiring sends an entry of type "start" to all involved Node Peers to start the processing of data entries. Thus, it starts the benchmark.

Guards:

1. **start (take)**: Entry triggering the start of the benchmark.
2. **iMap (read)**: Entry which contains all internal mappings received from I/O Peers. This is used to determine the addresses of the benchmarks Node Peers.

Actions:

1. **start**: Entry sent to all Node Peers in order to start the processing of data entries. Therefore, these entries start the benchmark.

4.5.3.4 W4: Stop Benchmark Wiring

After the benchmarks initialization time-frame a and the time-frame for forwarding data entries (the benchmarks run time) b, the wiring ends the benchmark.

Guards:

1. **stop (take)**: Entry triggering the stopping of the benchmark.
2. **iMap (read)**: Entry which contains all internal mappings received from I/O Peers. This is used to determine their addresses.

Actions:

1. **info**: Informs all I/O Peers that all their managed Node Peers have to be directed to stop the processing of data entries.
2. **stop**: Entry sent to all I/O Peers in order to stop the relaying of entries.

4.6 Summary

At the beginning of the chapter, the core functionalities of routing algorithms in unstructured P2P networks are identified. These functionalities are mapped to a generic abstract pattern, described in the Peer Model language. The composition of the frameworks top-level components not only supports arbitrary network topologies, it also provides a sophisticated separation of concerns and enables the distributed use of the framework as testbed.
Additionally, the very important performance aspect of a benchmarking framework is taken into consideration. Therefore, the framework is specifically designed to boost the

performance of the Peer Model to its fullest. In the rest of the chapter, the Peer Model language is used to specify the architecture and detailed concepts of the frameworks sub-components.

CHAPTER 5

Implementation Details

In this chapter, the implementation specifics of the framework are described.
The chapter is structured as follows.
After a general introduction of the programming language used, the Peer Model implementation and implemented extensions, the services of the framework are described. After that, the metrics, benchmarked by the framework, are introduced and the methodology of tracing is presented. Fruthermore, the benchmarks execution and configuration is described. Lastly, the frameworks limitations are discussed.

5.1 Peer Model Implementation & Extensions

The framework is implemented using the Peer Model, a coordination based programming model, presented in section 2.1. The Java 8 [59] implementation of the Peer Model is described in detail in [8]. On top of this Peer Model implementation, using the same programming language, the routing framework was implemented.
However, in order to enable the framework to handle huge amounts of entries, some extensions to the provided Peer Model implementation had to be made. When executing benchmarks with huge amounts of data entries, a significant drop in performance was experienced. To decrease the load of entries that have to be handled by the Peer Model implementation, the concept of nested entries was developed. Nested entries equal standard entries in the Peer Model implementation, except they allow to be stored in standard entries and other nested entries co- or app-data. In order to enable the use of nested entries in inter-peerspace-communication, contents of nested entries are stored in objects of type `FrameworkSerializedEntry` (figure 5.1) for the transmission. This is necessary, since the Peer Model implementation requires elements of an entries co- or app-data to be fully serializable and standard entries in the Peer Model implementation have attributes of type `java.util.Optional`, which are not serializable.
Entries of the same type, the same source, the same destination and with the same

transmission interference (e.g delay) that are sent at the same time are merged into a single container entry, which contains them as list of type `java.util.ArrayList` in its coordination data. Thus, instead of handling these entries separately, the Peer Model only has to cope with the container entry. However, before being processed, all entries are unpacked by a frameworks service and considered as separated. Without the usage of this nesting technique, it would be impossible for the routing framework to handle a realistic amount of data entries. Because only the contents of a nested entry are transmitted in the case of inter-peerspace-communication, a new object of type Entry is created on the receiving end. Thus, the internal Peer Model ID of the entry changes. However, this does not matter, since each data and routing communication entry has an additional ID, saved in the entries co-data, in the routing framework.

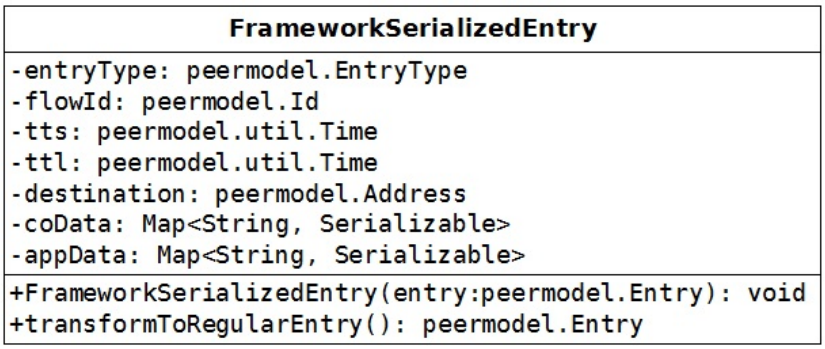

Figure 5.1: Class FrameworkSerializedEntry

Furthermore, in order to ease the dynamical sharing of peer address information during run time, objects of type `FrameworkPeerAddress` (figure 5.2) are used in the routing framework.

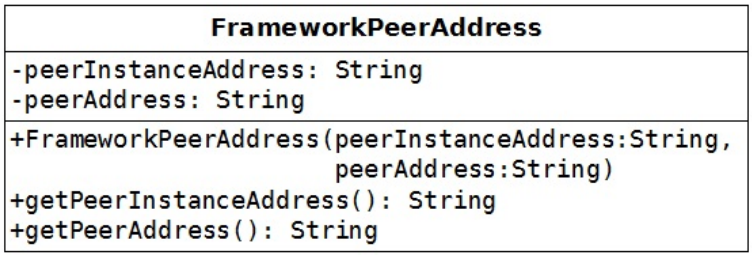

Figure 5.2: Class FrameworkPeerAddress

5.2 Services

While the wirings, their guards and actions, and the frameworks entry types are described in detail in chapter 4 , the services, as well as the interfaces provided for the implementation of routing algorithms, are presented in this section.

Since all services use a set of utility methods to extract entries from the collection passed by the wiring and to get specific co-data and app-data elements of entries, a super class `FrameworkPeerBaseService` exists. All services executed by the Node Peers (and all of its sub peers) wirings inherit from this super class. The utility methods provided by this super class are also very useful when implementing routing algorithms. Some of the services have a fixed implementation which is not subject to change. However, most of the services are abstract classes which provide several abstract methods. In order to implement a routing algorithm, either a provided default implementation is chosen, or a class which inherits from the respective abstract class has to be implemented. In the following, all services of the framework, the provided abstract methods to implement and the provided default implementations for each service are described.
It is important to note, that the class `peermodel.EntryBuilder` in the following subsections represents an unbuilt entry i.e. an entry which's contents can still be altered. Furthermore, the numbering of the services directly corresponds to the numbering of the wirings in chapter (e.g. service S1 of the Node Peer is the one executed by wiring W1 of the same peer).

5.2.1 Node Peer

5.2.1.1 S1, S3: TrafficDispatchService

This service is executed by the Data Dispatch Wiring and the Routing Dispatch Wiring. It provides the possibility to manipulate received entries before dispatching them. Furthermore, the service registers entries by adding an element of type "IN", the address of the Node Peer and the current synchronized time to the entries trace. For the detailed description of the tracing of data and routing communication entries, see subsection 5.3.1.

```
protected abstract void initDataEntry(EntryBuilder dataEntry);
```

Listing 5.1: Initialization of a data entry at its source

The `initDataEntry` method allows to add additional contextual information to a data entry at its source after it is placed in the PIC of the Node Peer by the Control Peer. Therefore, for each data entry, this method is executed exactly once at its source Node Peer.

```
protected abstract void manipulateDataEntry(EntryBuilder dataEntry);
```

Listing 5.2: Manipulation of a data entry before dispatching

The `manipulateDataEntry` method provides the possibility to manipulate a data entry when it is received and before it is dispatched for further processing by sub peers of the Node Peer. The method is not called if the Node Peer is the source of the data entry and the data entry has not been sent yet. Thus, either this method or the `initDataEntry` method is called before dispatching a data entry, but never both.

```
protected abstract void manipulateRtComEntry(EntryBuilder rtComEntry);
```

Listing 5.3: Manipulation of a routing communication entry before dispatching

The `manipulateRtComEntry` method allows the manipulation of routing communication entries before they are dispatched.

Default Implementation:

1. **NoInitAndManipulationTrafficDispatchService**: The default implementation of the `TrafficDispatchService` does neither add contextual information to data entries at the time of their initialization, nor manipulates entries before dispatching them.

5.2.1.2 S2: InitDispatchService

The services simply dispatches received entries of type "init" without altering them.

5.2.1.3 S4: StartService

The service creates an entry of type "started" which is then placed in the PIC of the Node Peer. Thus, after the execution of this service, the processing of data entries is started.

5.2.1.4 S5: StopService

The services simply dispatches a received entry of type "stop" without altering it.

5.2.2 Forwarding Peer

5.2.2.1 S1: RequestDecisionService

This is the service where, for each received data entry, a decision request is created.

```
protected abstract EntryBuilder createDecisionRequest(Entry dataEntry);
```

Listing 5.4: Creation of a decision request entry

The `createDecisionRequest` method is used to create the decision request entry. Based on the data entry, for which the request is created, additional context might be added to the request entries co- or app-data. However, the framework always adds the source and destination address of the data entry, as well as its unique identifier, to the requests app-data. Furthermore, to enable the mapping of decision request to data entries, the service always adds the corresponding data entry to the co-data of a created decision request.

Default Implementation:

1. **CopyDataEntryContextDecisionRequestService**: The default implementation of the `RequestDecisionService` extracts all contextual information contained in a data entry and adds it to the created decision request.

5.2.2.2 S2: ForwardDataService

The service is used to forward data entries via the Node Peers managing I/O Peer, based on made routing decisions. If a data entry has multiple next hops, the service clones the data entry accordingly. However, cloning of routing algorithm specific contents of a data entries co- or app-data has to be done manually in the `manipulateEntryBeforeSending` method.

```
protected abstract void manipulateEntryBeforeSending(
                                FrameworkPeerAddress thisNodePeerAddress,
                                Entry decisionAnswerEntry,
                                EntryBuilder dataEntry);
```

Listing 5.5: Manipulation of a data entry before sending

The `manipulateEntryBeforeSending` method allows to append additional contextual information to the data entries context (e.g. set the current Node Peers address as last hop of the data entry) and might be used for deep cloning of data entries.

Default Implementation:

1. **NoManipulationForwardDataService**: The default implementation of the `ForwardDataService` does not alter any data entry before sending.

5.2.2.3 S3: TerminateDataEntriesService

This service is used to send all unprocessed data entries to the Node Peers managing I/O Peer after the benchmarks run time has passed.

5.2.3 Routing Peer

5.2.3.1 S1: InitializationService

This service initializes all sub peers of a Routing Peer instance. The information needed is provided by the Control Peer.
It is important to note, that all methods available in this service for the implementation of routing algorithms have 2 address lists as parameters. The `neighborList` represents a list of unique identifiers of direct neighbors which's physical address is known. Therefore, direct communication (i.e. sending of entries) is possible. The `nodePeerAddresses` list contains a list of all unique identifiers of the other Node Peers. These identifiers can not be mapped to physical addresses (except those contained in the list of neighbors).

While these unique identifiers are often used to initialize algorithm specific structures such as routing tables, direct communication with non-neighbor nodes must not happen.

```
protected abstract EntryBuilder initializeRoutingInformationBase(
                        Entry initEntry,
                        FrameworkPeerAddress thisNodePeerAddress,
                        ArrayList<FrameworkPeerAddress> neighborList,
                        ArrayList<FrameworkPeerAddress> nodePeerAddresses);
```

Listing 5.6: Initialization of the routing information base

The `initializeRoutingInformationBase` method creates the entry that holds the Node Peers knowledge base. All structures needed by an algorithm can be created and initialized here. Note that the method must not return the value null.
Additionally, the framework always appends the address of the Node Peer, the neighbors address list, the list of identifiers of all the networks P2P nodes, the address of the benchmarks Statistics Peer and the address of the managing I/O Peer to the routing information base entries app-data.

```
protected abstract EntryBuilder initializeTerminationConditions(
                        Entry initEntry,
                        FrameworkPeerAddress thisNodePeerAddress,
                        ArrayList<FrameworkPeerAddress> neighborList,
                        ArrayList<FrameworkPeerAddress> nodePeerAddresses);
```

Listing 5.7: Initialization of the termination conditions

The `initializeTerminationConditions` method allows to establish an entry that holds information, which allows the Routing Information Peer to terminate received routing communication entries before processing them. If no entry for termination conditions (type “tCond”) is created, the framework creates and dispatches an empty one.

```
protected abstract ArrayList<EntryBuilder> createInitialRoutingSendEntries(
                        Entry initEntry,
                        FrameworkPeerAddress thisNodePeerAddress,
                        ArrayList<FrameworkPeerAddress> neighborList,
                        ArrayList<FrameworkPeerAddress> nodePeerAddresses);
```

Listing 5.8: Triggering of initial communication with other Node Peers

A set of entries of type “send” might be created to initiate communication with other Node Peers using the `createInitialRoutingSendEntries` method.

```
protected abstract ArrayList<EntryBuilder> createInitialUpdateEntries(
                        Entry initEntry,
                        FrameworkPeerAddress thisNodePeerAddress,
                        ArrayList<FrameworkPeerAddress> neighborList,
                        ArrayList<FrameworkPeerAddress> nodePeerAddresses);
```

Listing 5.9: Creation of initial entries to update the routing information base

If it fits the implemented routing algorithm, the `createInitialUpdateEntry` can be used to trigger initial updates of the routing information base after it is dispatched to the PIC of the Routing Information Peer.

Default Implementation:

1. **CopyInitEntryContextInitializationService**: The default implementation of the `InitializationService` does not add additional information to the routing information base.

5.2.4 Routing Decision Peer

5.2.4.1 S1: InformationRequestService

For each received decision request, a request for information is created in this service.

```
protected abstract EntryBuilder createInformationRequest(
                                                Entry decisionRequest);
```

Listing 5.10: Creation of an information request entry

The `createInformationRequest` method is used to create a information request based on a given decision request. This request is either a primitive one, without additional contextual information, or specifies in more detail which information is needed in order to answer the given decision request sufficiently. The implementation of the method must not return the null value. The framework always adds the source, the destination and the entries unique ID to the information requests application data. Furthermore, the information request always holds the corresponding decision request in its coordination data. Since the decision request holds the corresponding data entry in its co-data, the framework is always able to match corresponding requests to a data entry.

Default Implementation:

1. **NoAdditionalContextInformationRequestService**: The default implementation of the `InformationRequestService` does not append any additional contextual information to the routing information requests app-data.

5.2.4.2 S2: MakeDecisionService

This service represents the heart of each routing algorithm. It is were routing decisions are made. Before any method is called, the service extracts the corresponding decision request from the information answer entries co-data. After the execution of the service, the data entry is extracted from the decision request entries co-data and added to the decision answer entries co-data.

```
protected abstract boolean terminateDataEntry(
                                String entryID,
                                FrameworkPeerAddress thisNodePeerAddress,
                                FrameworkPeerAddress dataEntrySource,
                                FrameworkPeerAddress dataEntryDestination,
                                Entry decisionRequestEntry,
                                Entry informationAnswerEntry);
```

Listing 5.11: Termination of a data entry

The `terminateDataEntry` method allows to specify termination conditions for data entries (e.g. check if a maximum amount of hops is exceeded). Besides these routing algorithm specific termination conditions, the framework always checks if a data entry has reached its final destination. If that is the case, the data entry is terminated. If any of the termination conditions are met, the data entry is not forwarded any further, but sent to the Node Peers managing I/O Peer.

```
protected abstract void answerDecisionRequest(
                              String entryID,
                              FrameworkPeerAddress thisNodePeerAddress,
                              FrameworkPeerAddress dataEntrySource,
                              FrameworkPeerAddress dataEntryDestination,
                              Entry decisionRequestEntry,
                              Entry informationAnswerEntry
                              EntryBuilder decisionAnswerEntry
                              ArrayList<FrameworkPeerAddress> recipients);
```

Listing 5.12: Answering of a decision request

The `answerDecisionRequest` method is only called when a data entry is neither terminated nor has reached its destination. In this method, the decision process of a routing algorithm is implemented. The determined recipient(s) of the decision process are added to the parameter `recipients`. At least one recipient must be added to this list. Otherwise, the data entry will be terminated. Furthermore, if additional contextual information should be provided in the decision answer entry to the Forwarding Peer, it can be added to the app-data of the parameter `decisionAnswerEntry`.

```
protected abstract EntryBuilder updateInformationBase(
                                String entryID,
                                FrameworkPeerAddress thisNodePeerAddress,
                                FrameworkPeerAddress dataEntrySource,
                                FrameworkPeerAddress dataEntryDestination,
                                Entry decisionRequestEntry,
                                Entry informationAnswerEntry);
```

Listing 5.13: Creation of an asynchronous update request in the MakeDecisionService

Independently of the made routing decision (termination or forwarding of a data entry), the `updateInformationBase` method allows to create an update request entry to update the routing information base asynchronously.

Default Implementation:

1. **None**: Since this service is so specific to the implemented routing algorithm, the `MakeDecisionService` is the only service in the framework for which no default implementation is provided.

5.2.5 Routing Information Peer

5.2.5.1 S1.1 & S1.2: AnswerRequestService

As described in subsection 4.4.7, there exist 2 different wirings for answering information requests. However, the service executed by the wirings is always the same. The main task of the service is to process decision requests received from the Routing Decision Peer. After an information request has been processed, the services adds it to the information answer entries co-data. Thus, the framework is able to match corresponding information requests and answers.

```
protected abstract EntryBuilder createLockToken(
                                String entryID,
                                Entry routingInformationRequest,
                                Entry routingInformationBase,
                                FrameworkPeerAddress dataEntrySource,
                                FrameworkPeerAddress dataEntryDestination);
```

Listing 5.14: Creating a lock token for an information request

The `createLockToken` method is used to create a lock entry that blocks the information request from further processing (until the lock entry is removed). Note that this method must not be implemented (and therefore return the null value) if locks for the Routing Information Peer are not enabled in the benchmarks configuration. For detailed information regarding the benchmarks configuration, see section 5.5.

```
protected abstract EntryBuilder createInformationRequestAnswer(
                                String entryID,
                                Entry routingInformationRequest,
                                Entry routingInformationBase,
                                FrameworkPeerAddress dataEntrySource,
                                FrameworkPeerAddress dataEntryDestination);
```

Listing 5.15: Answering an information request

The `createInformationRequestAnswer` method is used to create information answer entries based on the received information request. The requested information is extracted from the routing information base. This method must not return the null value. The service always adds the address of the current Node Peer to the information answer entries app-data. Otherwise, the Routing Decision Peer would not be able to determine if a data entry has reached its destination. If wiring 1.2 is used (and therefore temporary blocking of information requests from being processed is enabled), the method

is only called if no lock was created for the corresponding information request by the `createLockToken` method.

```
protected abstract void updateRoutingInformationBase(
                                String entryID,
                                Entry routingInformationRequest,
                                EntryBuilder routingInformationBase,
                                FrameworkPeerAddress dataEntrySource,
                                FrameworkPeerAddress dataEntryDestination);
```

Listing 5.16: Updating the routing information base in the AnswerRequestService

If the service was able to process the information request, it is able to update the routing information base synchronously using the `updateRoutingInformationBase` method. This is done by manipulating the unbuilt entry provided by parameter `routingInformationBase`. Note that the update always happens after the information answer entry was created. Therefore, the update does not influence the answering of the corresponding routing information request.

```
protected abstract ArrayList<EntryBuilder> createSendEntries(
                                String entryID,
                                Entry routingInformationRequest,
                                EntryBuilder routingInformationBase,
                                FrameworkPeerAddress dataEntrySource,
                                FrameworkPeerAddress dataEntryDestination);
```

Listing 5.17: Creation of entries of type “send” in the AnswerRequestService

Independently of the information request being processed or blocked temporarily, the `createSendEntries` method allows to create multiple entries of type “send” to initiate communication with other Node Peers (e.g. to request information needed in order to be able to process a information request). If the method returns the null value, no communication is initiated.

Default Implementation:

1. **FullRoutingInformationBaseAnswerInformationRequestService**: The default implementation of the `AnswerRequestService` always provides all information contained in the routing information base to the Routing Decision Peer. It does not update the routing information base, initiate communication with other Node Peers or block information requests from being processed.

5.2.5.2 S2: SendRoutingInformationService

This service marks the place where the outgoing communication to other P2P nodes is realized. Any action of this service is based on contained information in the entries of type “send” (which trigger the execution of this service) and the state of the Node Peers routing information base. Before calling any of its methods, the service first checks if the currently processed “send” entry contains the value “true” in its co-data with key

"kill". If that is the case, the entry is terminated and sent to the Node Peers managing I/O Peer. This flag enables other wirings of the Routing Information Peer to correctly terminate entries, used for communication with other Node Peers.

```
protected abstract ArrayList<EntryBuilder> sendRoutingInformation(
                                Entry routingSendEntry,
                                Entry routingInformationBase,
                                FrameworkPeerAddress thisNodePeerAddress,
                                FrameworkPeerAddress ioPeerAddress,
                                FrameworkPeerAddress statisticsPeerAddress);
```

Listing 5.18: Creation of entries of type "rtCom"

Using the `sendRoutingInformation` method, entries can be sent to other Node Peers. Each of the unbuilt entries, contained in the list returned by this method, must have its recipient as instance of type `FrameworkPeerAddress` set in the entries co-data with key "recipient". Otherwise, the entry is dropped by the framework.
Furthermore, each of the returned entries must have its destination property set to the address of the Node Peers managing I/O Peer, provided in the input parameters of the method. If the destination property of a returned entry is not set, it is automatically placed in the PIC of the Routing Information Peer by the wiring.

```
protected abstract ArrayList<EntryBuilder> initiateAdditionalSending(
                                Entry routingSendEntry,
                                Entry routingInformationBase,
                                FrameworkPeerAddress thisNodePeerAddress);
```

Listing 5.19: Initiation of additional sending

All entries created by the `initiateAdditionalSending` method are set to type "send" and placed in the PIC of the Routing Information Peer. This allows to implement algorithm behavior like periodical sending of information or (periodical) spawning of additional intelligent agents.

Default Implementation:

1. **SendNoRoutingInfoService**: The `SendRoutingInformationService` default implementation does not implement any communication with other Node Peers.

5.2.5.3 S3: ReceiveRoutingInformationService

The service is used to process routing communication entries placed in the Routing Information Peers PIC.

```
protected abstract boolean terminateRtComEntry(
                                Entry receivedRtComEntry,
                                EntryBuilder terminationConditions,
                                Entry routingInformationBase,
                                FrameworkPeerAddress thisNodePeerAddress);
```

Listing 5.20: Termination of a received routing information entry

The `terminateRtComEntry` method determines if a received routing communication entry should be terminated. Therefore, it might uses the provided termination conditions and might also updates them in this method. If an entry is terminated, it is automatically sent to the Node Peers managing I/O Peer.

```
protected abstract ArrayList<EntryBuilder> createSendEntries(
                                Entry receivedRtComEntry,
                                Entry routingInformationBase,
                                FrameworkPeerAddress thisNodePeerAddress);
```

Listing 5.21: Creation of "send" entries in the ReceiveRoutingInformationService

The `createSendEntries` method is used to initiate communication with other Node Peers by creating entries of type "send". The method is not called if a received routing communication entry is terminated by the terminateRtComEntry service.

```
protected abstract EntryBuilder createUpdateRequest(
                                Entry receivedRtComEntry,
                                Entry oldTerminationConditions,
                                EntryBuilder updatedTerminationConditions,
                                Entry routingInformationBase,
                                FrameworkPeerAddress thisNodePeerAddress);
```

Listing 5.22: Creation of an update request in the ReceiveRoutingInformationService

If a received routing information entry is not terminated, the `createUpdateRequest` method allows to create a request to update the routing information base based on it.

Default Implementation:

1. **NoTerminationReceiveRoutingInfoService**: The default implementation of the `ReceiveRoutingInformationService` does not process any received routing information entries. Thus, no entries are terminated and neither entries of type "send" nor update request entries are created.

5.2.5.4 S4: UpdateInformationBaseService

The service processes update requests and therefore updates a Node Peers routing information base.

```
protected abstract void updateRoutingInformationBase(
                                Entry updateRequest,
                                EntryBuilder routingInformationBase,
                                FrameworkPeerAddress thisNodePeerAddress);
```

Listing 5.23: Updating the routing information base in UpdateInformationBaseService

The `updateRoutingInformationBase` method allows to alter the routing information base based on the received update request.

```
protected abstract ArrayList<EntryBuilder> createSendEntries(
                                Entry updateRequest,
                                Entry oldRoutingInformationBase,
                                Entry updatedRoutingInformationBase,
                                FrameworkPeerAddress thisNodePeerAddress);
```

Listing 5.24: Creation of entries of type "send" in the UpdateInformationBaseService

After the routing information base has been updated, the `createSendEntries` method can be used to initiate communication with other Node Peers based on the update. Therefore not only the update request is available, but also the routing information base in its states before and after the update. If the update entry should be terminated and therefore its contextual information sent to the Statistics Peer, it can be sent as entry of type "send" with the Boolean value "true" in its co-data with key "kill".

```
protected abstract ArrayList<EntryBuilder> createDeleteLockEntries(
                                Entry updateRequest,
                                Entry oldRoutingInformationBase,
                                Entry updatedRoutingInformationBase,
                                FrameworkPeerAddress thisNodePeerAddress);
```

Listing 5.25: Deletion of lock entries

The `createDeleteLockEntries` method is used to delete lock entries existing in the PIC of the Routing Information Peer. A lock entry is removed by creating a corresponding entry of type "delLo" with the same flow-ID. The method is only relevant when the creation of lock entries (blocking of information requests from being processed) is enabled in the benchmarks configuration. For detailed information regarding the benchmarks configuration, see section 5.5.

Default Implementation:

1. **NoUpdateInformationBaseService**: The default implementation of the `UpdateInformationBaseService` does not process any received routing update requests.

5.2.5.5 S5: DeleteLockService

This service simply consumes a lock entry and the corresponding entry which triggers its deletion.

5.2.6 I/O Peer

5.2.6.1 S1, S2 SendEntriesService

The wirings for relaying data entries and routing communication entries (W1 and W2) both execute this service. Based on the entries source and destination, the network links attributes are read from the I/O Peers external mapping. After optional transmission interference is applied, entries are sent either directly to their destination or to their destinations managing I/O Peer, depending on which Peer Space the destination is located. Additionally, as the I/O Peer marks the outgoing port for a Node Peer, the service adds an element of type "OUT", the address of the sending Node Peer and the current synchronized time to the entries traces. If the recipient of processed entries is the Statistics Peer, the service extracts the time / location traces from them, merges them in a list and sends them to the Statistics Peer. For a detailed description of the tracing of data and routing communication entries, see subsection 5.3.1.

```
protected abstract void applyNetworkTransmissionProperties(
                                                    EntryBuilder entry,
                                                    PeerLink targetPeerLink);
```

Listing 5.26: Deletion of lock entries

The `applyNetworkTransmissionProperties` method allows to apply transmission interference (e.g. delay, dropping of entries) before an entry is relayed. The parameter `targetPeerLink` contains the network links attributes and is described in detail in subsection 5.2.8.

Default Implementation:

1. **DelaySendEntriesService**: The `SendEntriesService` default implementation applies the delay, specified in the link between source and destination, via an entries TTS before it is sent.

5.2.6.2 S3, S4: ReceiveEntriesService

The wiring W3, which handles received data entries (type "outD"), and W4, that handles received routing communication entries (type "outR"), execute the same service. Thus, the service does not distinguish between entries of these types. It simply looks up the destination in the I/O Peers internal mapping and relays an entry accordingly.

5.2.6.3 S5: SendRegistrationService

The service simply builds an entry that includes the I/O Peers address and its internal mapping (includes the addresses of all managed Node Peers). This entry is then sent to the Control Peer for the purpose of registration.

5.2.6.4 S6: StopNodePeersService

The service simply clones the received entry of type "stop" to ensure that all of the I/O Peers managed Node Peers get informed that the benchmark has ended.

5.2.6.5 S7: CleanupService

After its run time, the benchmark is stopped. After that, the `CleanupService` extracts the time / location traces from all remaining data and routing communication related entries (types "data", "outD", "rtCom" and "outR"), merges them to a single list and sends them to the Statistics Peer. This ensures that all entries are included in the calculation of statistics. However, if a routing communication entry (type "rtCom" or "outR") does not contain an element in its trace, the service implicitly detects that it was not part of any communication between peers and therefore ignores it. For a detailed description of the tracing of data and routing communication entries, see subsection 5.3.1.

5.2.7 Statistics Peer

In the Statistics Peer, relevant information is gathered, the statistics of the benchmark are calculated and outputted after the benchmark has ended. The central entity, in which calculated metrics (described in section 5.3) are stored, is the `Metrics` object shown in figure 5.3.

5.2.7.1 S1: StatCalculationService

In this service, the benchmarks statistics are calculated. For each uniquely received data entry (determined by an entries ID), its trace is stored as list of `TraceRecord` objects. The class `TraceRecord` is described in detail in subsection 5.3.1. If copies of an unique data entry are received, only the fastest one is added to the list of unique data entries, the rest is stored as duplicates. For duplicates and dropped data entries only three metrics are stored in objects of type `StatisticsRecord`: the hop count, the time in milliseconds it took the entry from source to destination and an indicator if the taken path contains a loop. The class `StatisticsRecord` is shown in figure 5.4. For routing communication entries, only the amount of received entries and their accumulated hop count are tracked.

5.2.7.2 S2: StatisticsOutputService

This service is used to output the values of the calculated statistics. Therefore, the method `outputStatistics` is used.

```
protected abstract void outputStatistics(Metrics metrics,
                                         String outputPath);
```

Listing 5.27: Output of calculated statistics

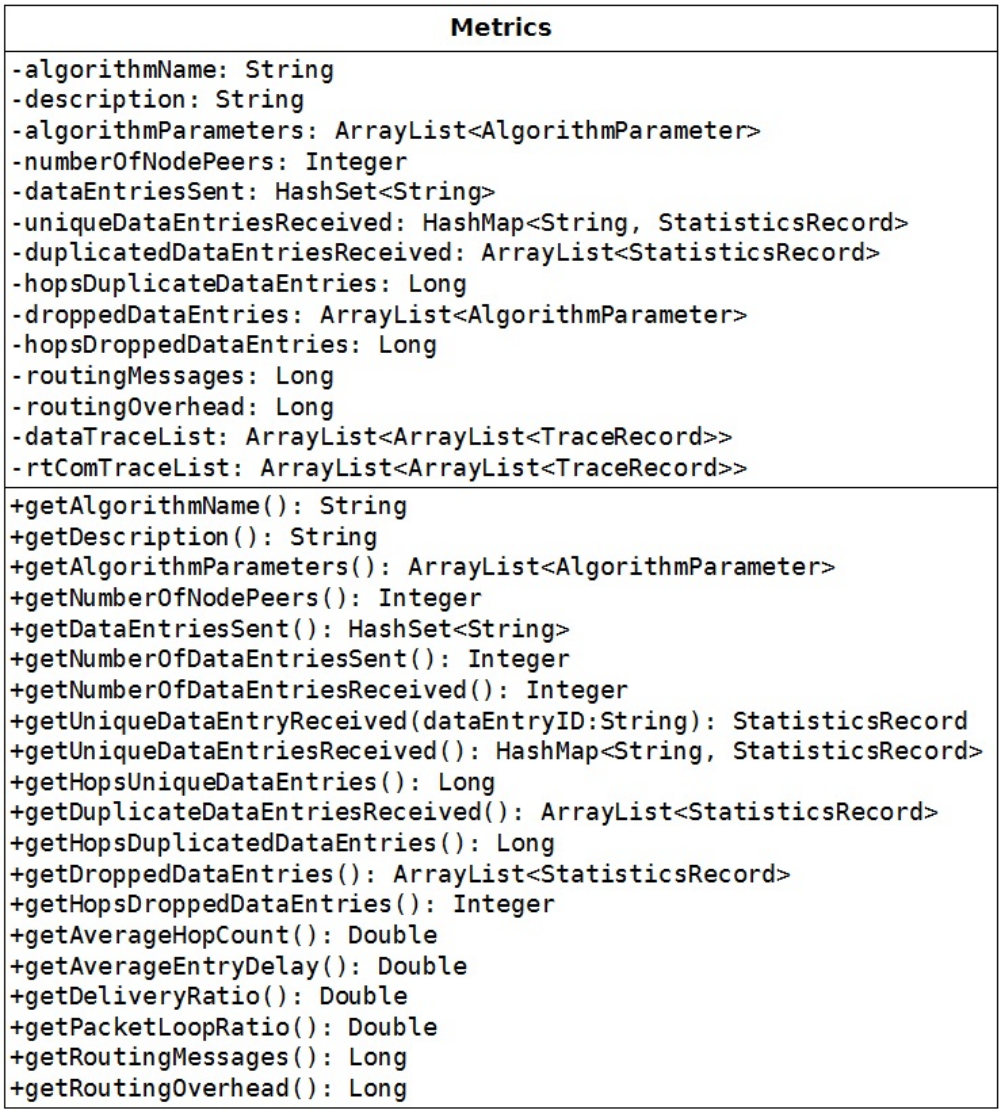

Figure 5.3: Class Metrics. For reasons of simplicity, setter methods are ignored in this figure.

The parameter `outputPath` is set in the frameworks configuration. Note that it is possible to set the implemented routing algorithms parameters as a list of objects of type `AlgorithmParameter` (shown in figure 5.5) for a `Metrics` instance. However, this is not done directly, but can be set via the static method `setAlgorithmParameters` of class `AlgorithmConfigProvider`. This provides the opportunity to display the algorithms parameters in the chosen output format.

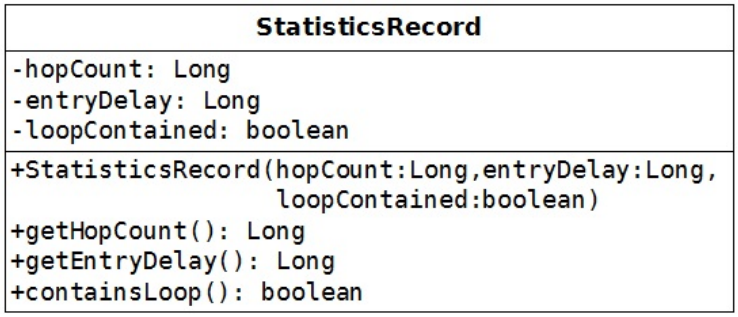

Figure 5.4: Class StatisticsRecord

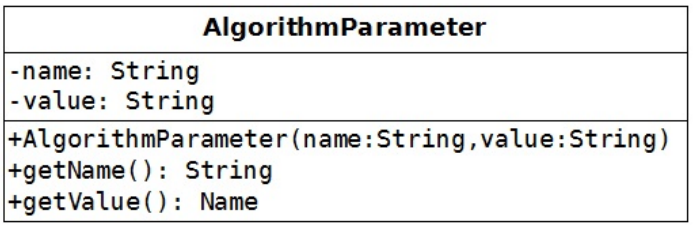

Figure 5.5: Class AlgorithmParameter

Default Implementations:

1. **CommandLineStatisticsOutputService**: Visualization through formatted output of the benchmarks results on the standard output stream. Generally, this is the command-line.

2. **XLSStatisticsOutputService**: The output of benchmarks is saved in a single sheet of an Excel file (type `XLS`).

3. **XLSSheetStatisticsOutputService**: Provides additional features compared to the `XLSStatisticsOutputService`. In addition to the listed results, the average of each metric is calculated automatically. When the benchmarks description changes, the results are stored in a new sheet. Furthermore, the created file contains an overview sheet, which shows the average results for all benchmarked configurations (assuming that different configurations have different benchmark descriptions).

4. **XLSSheetHistoryStatisticsOutputService**: Generally, this default implementation operates like the `XLSSheetStatisticsOutputService`, but copies all results and saves them into a new file when the benchmarks description changes. This ensures that if the specified XLS file gets corrupted, not all results are lost.

All default implementations that output results in the XLS format are implemented using the jExcel API [60]. It is important to note, that each Excel file should be used for a single algorithm only.

5.2.8 Control Peer

5.2.8.1 S1 RegisterNodePeerService

This service registers Node Peers and their managing I/O Peer by adding their addresses and relation to the collection of internal mappings held by the Control Peer.

5.2.8.2 S2 InitializeBenchmarkService

The service is responsible to build the benchmarks topology and initialize all of the frameworks topology dependent structures. After the topology is created, the involved I/O Peers internal and external mappings are generated. Then, the initialization entries for all Node Peers are created. Furthermore, the configured amount of data entries, each data entry with a random source and destination, is generated. Additionally, the entry to start the benchmark, the entry to stop the benchmark, the timer entries and the entry which triggers the output of the benchmark are created. Each with the configured TTS set.

```
protected abstract ArrayList<TopologyLink> createTopology(
                        ArrayList<FrameworkPeerAddress> nodePeerAddresses,
                        Integer minDelayInMs,
                        Integer maxDelayInMs);
```

Listing 5.28: Creation of the benchmarks topology

The `createTopology` method is used to create the topology of the P2P network on which the benchmark is executed. It takes the addresses of all involved Node Peers and the configured interval boundaries for the transmission delay of the networks links as input. The method generates a list of the P2P networks links, implemented as instances of class `TopologyLink` (shown in figure 5.6). A link connects two Node Peers. Transmission of data over a link is delayed by a defined amount.

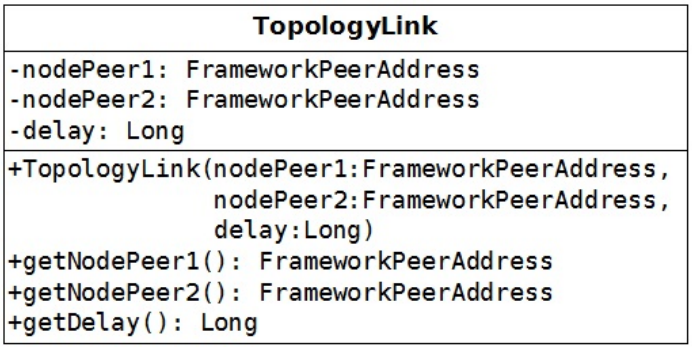

Figure 5.6: Class TopologyLink

Default Implementations:

1. **InitBenchmarkScaleFreeTopology**: The network topology is generated using the algorithm of the Barabási-Albert model [3].

   A fundamental attribute of the algorithm is, that for large generated graphs, their vertices degrees are power-law distributed. Thus, for large networks, the probability P_1 that a vertex in the generated graph has the degree k follows $P_1(k) \sim k^{-\gamma}$ with $\gamma = 2.9 \pm 0.1$ [3].

   The Barabási-Albert model algorithm starts with an initial set of m_0 connected vertices. Then, one at a time, till the desired size of the network is reached, vertices are added. Each newly added vertex is connected to $m \leqslant m_0$ already existing vertices in the network graph. However, the m connections are chosen based on a roulette wheel selection with probability

   $$P_2 = \frac{k_i}{\sum_j k_j} \tag{5.1}$$

   Thus, newly added vertices are more likely to be connected to existing vertices with a high degree k_i. Based on [23] and [7], $m = m_0 = 2$ is chosen for this implementation of the algorithm. An example instance of a topology created by the Barabási-Albert model algorithm is $G_{ex4} = (V_{ex4}, E_{ex4})$ with $V_{ex4} = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$ and $E_{ex4} = \{(x_1, x_2), (x_1, x_3), (x_2, x_3), (x_2, x_4), (x_3, x_4), (x_1, x_5), (x_3, x_5), (x_1, x_6), (x_3, x_6), (x_1, x_7), (x_6, x_7)\}$. It is shown in figure 5.7.

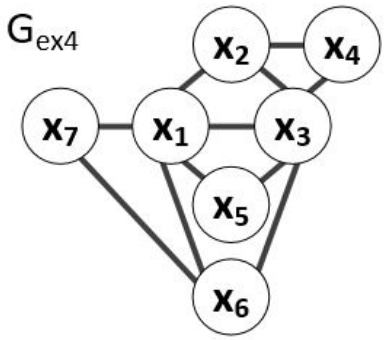

Figure 5.7: Example of a scale-free network instance generated by the Barabási-Albert model algorithm.

5.2.8.3 S3: StartBenchmarkService

Creates an entry of type "start" for each registered Node Peer.

5.2.8.4 S4: StopBenchmarkService

The service creates $k \geqslant 1$ entries of type "stop". k is the number of registered I/O Peers, to which the cloned entries are then sent.

5.3 Tracing & Output

The primary task of the framework is to benchmark routing algorithms. Thus, metrics need to be defined that allow the evaluation of benchmarked algorithms. Furthermore, the correct collection of relevant data has to be ensured. This section describes the frameworks time synchronization and how entires are traced in the framework. Moreover, the metrics contained in the frameworks provided output are discussed.

5.3.1 Entry Tracing

In order to enable the framework to calculate any metric values, information about data and routing communication entries has to be collected. This is done using a time / location trace list for each entry. Each record of the trace list is represented by an instance of class `TraceRecord`, shown in figure 5.8. A trace record contains the direction (inbound or outbound), the time (in millisecond between the current time and January 1, 1970 00:00 UTC) and the location (address of the Node Peer). It is added to the trace list when a Node Peer dispatches a received data or routing communication entry and when an I/O Peer relays one of those entries. In figure 5.9, an example trace list is shown.

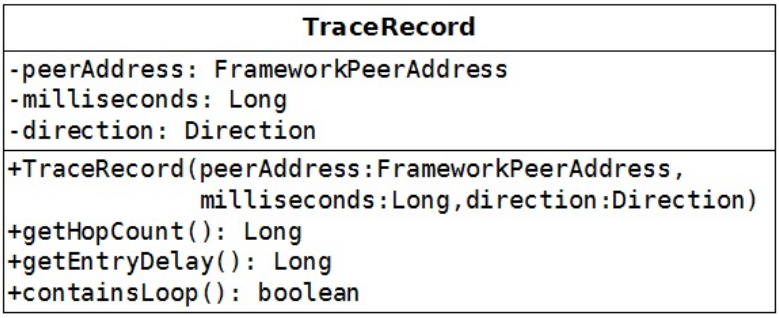

Figure 5.8: Class TraceRecord

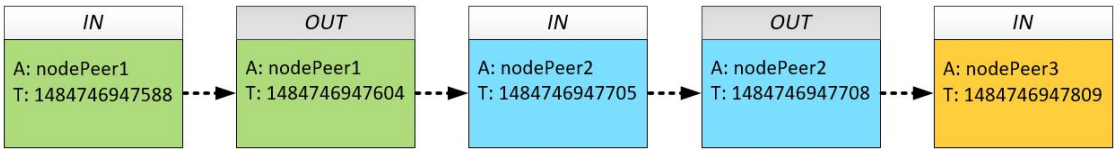

Figure 5.9: Example of a full entry trace list. The entry, originated by nodePeer1, visited exactly one intermediate node before reaching its destination.

5.3.2 Time Synchronization

As described above, trace records are added locally to an entries trace list at visited Node Peers and their managing I/O Peers. However, if multiple Node Peer Spaces are involved in the benchmark, their clocks are not necessarily synchronized. This is especially the

case, when Node Peer Spaces are run on different physical hosts. Thus, asynchronous clocks on Node Peer Spaces might lead to falsified benchmark results.
If the framework is not operated in offline mode, it synchronizes the clock of each Node Peer Space with a NTP-server using the Simple Network Time Protocol (SNTP) [31]. The address of the server is configurable.
At start time, at each Node Peer Space, the framework determines the offset in milliseconds to the NTP-servers clock five times and uses the calculated average. The synchronized clock is accessed via the method `currentTimeMillis()` of the frameworks class `SynchedFrameworkClock`. It returns the sum of the difference between the current local time at a Node Peer Space and January 1, 1970 00:00 UTC in milliseconds and the calculated average offset. It is important to stress that this method should always be used instead of the identically named method of class `java.lang.System` when implementing routing algorithms. For the SNTP part of the implementation, the Apache Commons Net library [54] was used.

5.3.3 Benchmark Output

The output, provided by the framework after a benchmark has ended, is shown in tables 5.1 and 5.2.

Metric	Description
number of Node Peers	Size of the benchmarks network instance.
unique data packets sent	Amount of unique data packets that were sent from a random source to a random destination in the benchmark. Each of these data packets has an unique id assigned.
overall delivered data packets	Amount of data packets delivered. Includes uniquely delivered data packets and duplicates.
unique delivered data packets	Amount of unique data packets delivered. If duplicates are delivered, the fastest is considered to be the unique one, the rest is interpreted as duplicates.
unique delivered data packet hops	Accumulated amount of hops delivered unique data packets have made.
delivered data packet duplicates	Amount of data packet duplicates delivered.
delivered data packet duplicate hops	Accumulated amount of hops delivered data packet duplicates have made.

Table 5.1: Metrics outputted by the framework (part 1)

Metric	Description
dropped data packets	Amount of data packets dropped on their way from their source to their destination.
dropped data packets hops	Accumulated amount of hops dropped data packets have made.
data packet delivery ratio	Shows the ratio of delivered data packets. $$delivery\ ratio = \frac{unique\ data\ packets\ sent}{unique\ delivered\ data\ packets} \quad (5.2)$$ Based on [48].
average data packet hop count	Average amount of hops delivered unique data packets have made. Based on [45].
average data packet delay	Average delay of delivered unique data packets from source to destination. Based on [48].
data packet loop ratio	Shows the ratio of delivered unique data packet paths containing loops. A data packets path contains a loop if a network node has been visited more than once on the path from source to destination. $$loop\ ratio = \frac{unique\ data\ packets\ containing\ loops}{unique\ delivered\ data\ packets} \quad (5.3)$$ Based on [48].
routing overhead	Routing overhead of the benchmarked routing algorithm. $$\begin{aligned} &routing\ overhead = \\ &delivered\ data\ packet\ duplicate\ hops + \\ &dropped\ data\ packet\ hops + \\ &routing\ communication\ packet\ hops \end{aligned} \quad (5.4)$$ Based on [48].
routing overhead messages	Amount of routing overhead packets sent. $$\begin{aligned} &routing\ overhead\ messages = \\ &delivered\ data\ packet\ duplicates + \\ &dropped\ data\ packets + \\ &routing\ communication\ packets \end{aligned} \quad (5.5)$$ Based on [48].

Table 5.2: Metrics outputted by the framework (part 2)

5.4 Framework execution

The framework is provided in two executable jar-files:

- `peermodel-routing-framework-controlpeer.jar`
- `peermodel-routing-framework-nodepeer.jar`

The former is used to create and run a new Control Peer Space instance, the latter a new Node Peer Space instance. Table 5.3 describes the parameters which must be provided for executing the `peermodel-routing-framework-controlpeer.jar`.
Table 5.4 those for the `peermodel-routing-framework-nodepeer.jar` executable.

Position	Name	Description
1	Control Peer Config File	Path to the Control Peer Space configuration `properties` file.

Table 5.3: `peermodel-routing-framework-controlpeer.jar` parameters

Position	Name	Description
1	Node Peer config file	Path to the Node Peer Space configuration `properties` file.
2	Control Peer config file	Path to the Control Peer Space configuration `properties` file. Needed since it contains the benchmarks initialization and run time interval.
3	Node Peer count	Amount of Node Peers to be spawned at this Node Peer instance.

Table 5.4: `peermodel-routing-framework-nodepeer.jar` parameters

The `peermodel-routing-framework-controlpeer.jar` file must not be executed more than once per benchmark. However, since multiple Node Peer Spaces may be involved in a benchmark, the amount of executions of the `peermodel-routing-framework-nodepeer.jar` file is not restricted.
Furthermore, the order of executing the jar-files is important. The Control Peer Space must always be created before the Node Peer Spaces.

5.5 Framework Configuration

The framework is configured using two `properties` files:

- `controlpeer_config.properties`, which is used for the configuration of the Control Peer Space. Its configurable parameters and, if available, their default values, are shown in tables 5.5.
- `nodepeer_config.properties`, which is used for the configuration of the Node Peer Space. Its configurable parameters and, if available, their default values, are shown in tables 5.6, 5.7 and 5.8.

The goal is to provide extensive configurability while keeping it as simple as possible. Therefore, a lot of parameters have default values, which fit most scenarios.

Parameter	Description
controlPeerPeerSpace InstanceAddress	Defines the address of the Control Peer Space.
controlPeerAddress	Defines the address of the Control Peer.
benchmarkStartTime	Defines the length of the benchmarks initialization interval in milliseconds.
benchmarkRunTime	Defines the length of the benchmarks run interval in milliseconds.
numberOfNodePeers	Defines the network size of the benchmarks network instance.
numberOfDataEntries	Defines the amount of unique data entries to be sent in the benchmark.
delayLowerBoundInMs	Defines the lower limit for the randomly chosen delays for transmission on a network link.
delayUpperBoundInMs	Defines the upper limit for the randomly chosen delays for transmission on a network link.
initBenchmarkService	Chooses the used implementation of the Control Peers `InitializeBenchmarkService`.
statOutputService	Chooses the used implementation of the Statistic Peers `StatOutputService`. **Default:** `CommandLineStatisticsOutputService`
algorithmName	Name of the benchmarked algorithm. Might be used by the Statistic Peers `StatOutputService` implementation.
description	Description of the benchmark. Might be used by the Statistic Peers `StatOutputService` implementation.
outputPath	Output path to which the result file should be saved. Might be used by the Statistic Peers `StatOutputService` implementation.

Table 5.5: Parameters for the configuration of the Control Peer Space

Parameter	Description
controlPeerPeerSpace InstanceAddress	The address of the Control Peer Space.
controlPeerAddress	The address of the Control Peer where the Node Peers register.
offlineMode	Deactivates/Activates SNTP synchronization. **Default:** `false`
ntpHost	NTP server used for time synchronization. Must be set if offline mode is not activated.
enableRouting InformationPeerLocks	Enables temporary blocking of information requests by lock entries. **Default:** `false`
entryTypePriority	Specifies the priority of types of entries when processing them in the framework. (Available modes are NONE (shared queues), DATA or RTCOM). **Default:** `NONE`
dataDispatchWiring MaxEntries	Maximum amount of data entries dispatched by a Node Peer with a single service execution. **Default:** `999999`
routingDispatchWiring MaxEntries	Maximum amount of routing communication entries dispatched by a Node Peer with a single service execution. **Default:** `999999`
requestChain MaxEntries	Maximum amount of entries handled with a single service execution by all wirings involved in the request/answer chain to determine the next hop(s) of a data entry. **Default:** `999999`
sendRouting InformationMaxEntries	Maximum amount of received routing communication entries handled by the `Send Routing Information Wiring` with a single service execution. **Default:** `999999`
handleReceived MaxRoutingEntries	Maximum amount of received routing communication entries handled by the `Receive Routing Information Wiring` with a single service execution. **Default:** `999999`
trafficDispatchService	The routing algorithms implementation of the `TrafficDispatchService`. **Default:** `NoInitAndManipulationTrafficDispatchService`
forwardDataService	The routing algorithms implementation of the `ForwardDataService`. **Default:** `NoManipulationForwardDataService`

Table 5.6: Parameters for the configuration of the Node Peer Space (part 1)

Parameter	Description
decisionRequestService	The routing algorithms implementation of the `DecisionRequestService`. **Default:** `CopyDataEntryContextDecisionRequestService`
initializationService	The routing algorithms implementation of the `InitializationService`. **Default:** `CopyInitEntryContextInitializationService`
informationRequest Service	The routing algorithms implementation of the `InformationRequestService`. **Default:** `NoAdditionalContext InformationRequestService`
makeDecisionService	The routing algorithms implementation of the `MakeDecisionService`.
answerRequestService	The routing algorithms implementation of the `AnswerRequestService`. **Default:** `FullRoutingInformationBaseAnswer InformationRequestService`
receiveRoutingInfo Service	The routing algorithms implementation of the `ReceiveRoutingInformationService`. **Default:** `NoTerminationReceiveRoutingInfoService`
sendRouting InformationService	The routing algorithms implementation of the `SendRoutingInformationService`. **Default:** `SendNoRoutingInfoService`
updateInformationBase Service	The routing algorithms implementation of the `UpdateInformationBaseService`. **Default:** `NoUpdateInformationBaseService`

Table 5.7: Parameters for the configuration of the Node Peer Space (part 2)

Parameter	Description
sendWiringMaxEntries	Maximum amount of entries handled by the `Send Data Wiring` and `Send Routing Communication Wiring` with a single service execution. (I/O Peer) **Default:** `999999`
receiveWiring MaxEntries	Maximum amount of entries handled by the `Receive Data Entries Wiring` and `Receive Routing Entries Wiring` with a single service execution. (I/O Peer) **Default:** `999999`
sendEntriesService	The implementation of the I/O Peers `SendEntriesService`. **Default:** `DelaySendEntriesService`

Table 5.8: Parameters for the configuration of the Node Peer Space (part 3)

5.6 Limitations

Since the proposed framework is built on top of the Peer Model Java implementation [8], it is affected by its limitations.
The Peer Model implementation does not allow a service to determine the amount of entries in its peers containers. Thus, currently, routing algorithms implemented in the framework are not able to include queue sizes in their calculations.
Furthermore, entries are not consumed by wirings in a specified order, but randomly. However, this limitation is not as critical for the framework, because services take multiple entries at once for processing and the same service may also runs parallel multiple times at the same time.
Additionally, the use of priority queues is limited in the framework. Only one of the priority modes for entry processing (data entries or routing communication entries) can be chosen in a benchmarks configuration. It is currently not possible to implement dynamic priority modes (e.g. routing communication entries share priority queues with data entries under certain circumstances but are otherwise prioritized). This results from the fact that priority queues are built statically at the beginning of the benchmark by adding guards to wirings. The wirings can't be changed during run time.
Furthermore, when implementing routing algorithms in the framework, one has to take in mind that the method `copy` of class `peermodel.EntryBuilder` of the Peer Model implementation does not provide deep clones of an entry. Thus, routing algorithm specific co- and app data content has to be cloned manually if needed. Otherwise, this behavior could result in concurrency problems.

5.7 Summary

In this chapter, made extensions to the Peer Model Java implementation [8] are described. They allow to meet the strict performance needs of a benchmarking framework and boost the performance to its fullest.
Furthermore, the services and their interfaces, used for the implementation of routing algorithms and the creation of benchmarks are discussed. While still being generic for the implementation of diverse routing algorithms, they are simple and specific enough to be easily picked up by new developers. Furthermore, the provided default implementations, minimize the implementation effort.
The same concept applies to the execution and the configuration of the framework. While enable the in-depth configurability of the framework for advanced users, default values, suitable for most scenarios, are set to the majority of configuration parameters. Therefore, new users can directly focus on the benchmarking of routing algorithms instead of fiddling with the benchmarks configuration.
There are still some limiting factors to the framework that are described at the end of the chapter. However, some of them will most likely fade when the development of the Peer Model goes on.

CHAPTER 6

Evaluation

In this chapter, the BeeNet and SlimeMoldNet routing algorithm adaptions are analyzed and compared to five prominent intelligent and non-intelligent routing algorithms.

6.1 Benchmark Methodology

For the comparison, the routing algorithms Gnutella Flooding, k-Random Walker, AntNet, BeeHive and Physarum polycephalum are benchmarked alongside the BeeNet and SMNet routing algorithms. Each of this five routing algorithms is briefly described in subsection 2.3.3. The BeeNet and SMNet are presented in 3.2 and 3.3.

All benchmarks are carried out in the Google Compute Engine cloud infrastructure [56]. More specifically, a "n1-standard-16" instance is used, on which the Ubuntu 16.04 LTS operating system is run. The instance includes 16 vCPUs and 60 GB RAM. According to [57], a vCPU equals a hardware thread of a 2.6 GHz Intel Xeon E5, a 2.5 GHz Intel Xeon E5 v2, a 2.3 GHz Intel Xeon E5 v3 or a 2.2 GHz Intel Xeon E5 v4 CPU.

The general benchmark parameters are:

- **Topology:** The topology of the unstructured P2P network is a scale-free network generated using the algorithm of the Barabási-Albert model [3]. For this cause, the default implementation `InitBenchmarkScaleFreeTopology` of the frameworks abstract class `InitializeBenchmarkService` is used. Implementation specifics and the Barabási-Albert model are described in 5.2.8.2.

- **Data packets:** Data packets are placed at a random source and have a random destination. The amount of data packets is varied in three levels: low (L_d), medium (M_d) and high (H_d) where $L_d = 100$, $M_d = 500$ and $H_d = 1000$ data entries.

- **P2P network nodes:** The amount of P2P network nodes is also varied in three levels: low (L_n), medium (M_n), high (H_n) where $L_n = 50$, $M_n = 100$, $H_n = 200$. Furthermore, it is important to note that there is no fluctuation of node participation. Thus, no churn is simulated.

- **Transmission delay:** The transmission delay for exchanging packets between neighbors is set to 100 ms for all links of the P2P network.

- **Benchmark initialization time:** The amount of time nodes have to initialize their routing related structures before data packets are sent. The time interval $init\_time$ is dependent on the level of the amount of participating P2P node peers. More specifically, $init\_time(L_n) = 120000\ ms$, $init\_time(M_n) = 240000\ ms$, $init\_time(H_n) = 480000\ ms$.

- **Benchmark run time:** The amount of time in which data entries have to be delivered. If this time interval has passed, not delivered data packets are terminated. Similar to the benchmarks initialization time, this time interval run_time is also dependent on the level of the amount of participating P2P node peers. $run\_time(L_n) = 30000\ ms$, $run\_time(M_n) = 60000\ ms$, $run\_time(H_n) = 120000\ ms$.

- **Benchmark execution:** For each configuration of an algorithm, 10 benchmarks are executed. This is due to the non-deterministic nature of the benchmarked routing algorithms. In the result data the average metric values are reported for each configuration.

For the rest of the frameworks parameters, the suggested default values, described in 5.5, are used.

The metrics used for evaluation and analysis are **data packet delivery ratio**, **average data packet delay**, **average data packet hop count** and **routing overhead messages**. They are described in detail in 5.3.3.

6.2 Sensitivity Analysis

The goal of the Sensitivity Analysis is to find the optimal parameter values of all benchmarked algorithms for each configuration. The analysis is performed on the value range of parameters recommended by the algorithms authors or determined in preliminary benchmarks. Thus, parameters are only subject of the Sensitivity Analysis, if no fixed parameter value is recommended by an algorithms author or a recommended value showed to be very suboptimal in the benchmark scenario. More specifically, for all of a routing algorithms parameters, that are subject of the Sensitivity Analysis, the optimal value for all combinations of the three network sizes (L_n, M_n, H_n) and the three data packet amount levels (L_d, M_d, H_d) is determined based on the metrics stated in 6.1. The determined optimal parameter values are used in the competitive benchmarks.

All parameters of the AntNet algorithm, described in section 2.3.3.3, routing algorithm are chosen based on the recommendations in [12]. They showed good results in the preliminary benchmarks. Therefore, no Sensitivity Analysis is necessary. However, it is important to note, that due to the frameworks current limitations regarding the consideration of queue sizes, discussed in 5.6, the parameter for weighting the output queue size when selecting the next hop α, has to be chosen $\alpha = 0$. The parameter values used for all competitive benchmarks are shown in table 6.1.

Parameter	**Value**	**Source**
$antSpawnInterval$	300 ms	[12]
TTL	15000 ms	
α	0	limitations 5.6
η	5	[12]
c	0.3	
c_1	0.7	
c_2	0.3	
γ	0.7	
a	4	

Table 6.1: AntNet parameter values, used for all competitive benchmarks

For the BeeHive routing algorithm, described in section 2.3.3.5, the optimal `beeSpawnInterval` showed in preliminary benchmarks to be highly dependent on the network size. Therefore, this parameter was subject of the Sensitivity Analysis. However, all other parameter values, used for competitive benchmarks, are based on the recommendation in [18].
The parameter values before the Sensitivity Analysis are shown in table 6.2, while its results are shown in table 6.3.
Interestingly, the `beeSpawnInterval` shows to be resistant to the chosen data packet amount level but very sensitive to the network size. The bigger the size of the network, the more the continuous flooding of bees congests the P2P networks traffic.

Parameter	**Value (Range)**	**Source**
$beeSpawnInterval$	3000 ms, 6000 ms, 9000 ms, 12000 ms, 15000 ms	preliminary benchmarks
$initializationInterval$	30000 ms	[18]
$shortDistanceLimit$	7	
$longDistanceLimit$	40	

Table 6.2: BeeHives parameter values before the Sensitivity Analysis

nodes	packets	beeSpawnInterval
50	100	9000 ms
50	500	
50	1000	
100	100	
100	500	
100	1000	
200	100	15000 ms
200	500	
200	1000	

Table 6.3: BeeHive Sensitivity Analysis results

For the Physarum polycephalum routing algorithm, described in section 2.3.3.4, both parameters, ϵ and the `agentSpawnInterval` are subject of the Sensitivity Analysis. The parameter values before the analysis are shown in table 6.4, while the results are shown in table 6.5. The analysis shows that the algorithm performs best with a learning factor ϵ on the upper bound of the recommended value range and a rather small `agentSpawnInterval`.

Parameter	Value (Range)	Source
$agentSpawnInterval$	300 ms, 1000 ms, 3000 ms	preliminary benchmarks
ϵ	0.025, 0.05, 0.1	[19]

Table 6.4: Physarum polycephalum algorithms parameter values before the Sensitivity Analysis

nodes	packets	agentSpawnInterval	ϵ
50	100	300 ms	0.1
50	500		
50	1000		
100	100		
100	500		
100	1000		
200	100		
200	500		
200	1000		

Table 6.5: Physarum polycephalum routing algorithm Sensitivity Analysis results

The Sensitivity Analysis is performed on Gnutella Floodings only parameter, TTL. The TTL value range is shown in 6.6, the results of the analysis in table 6.7. A TTL-value of 7 shows to be sufficient for delivering data packets reliably. For the description of the algorithm, see section 2.3.3.1.

Parameter	**Value (Range)**	**Source**
TTL	7, 8, 9	preliminary benchmarks

Table 6.6: Gnutella Flooding parameter values before the Sensitivity Analysis

nodes	**packets**	**TTL**
50	100	7
50	500	
50	1000	
100	100	
100	500	
100	1000	
200	100	
200	500	
200	1000	

Table 6.7: Gnutella Flooding Sensitivity Analysis results

The Sensitivity Analysis is performed on both of k-Random Walkers parameters. The algorithm in described in 2.3.3.2. While the parameter value range for `walkerAmount` is based on the recommendation in [30], the value range walkers TTL resulted from preliminary benchmarks. The parameter values before the analysis are shown in 6.8, its results in table 6.9. Due to the random nature of the algorithm, all values below the upper bound of value ranges show insufficient efficiency regarding the delivery ratio.

Parameter	**Value (Range)**	**Source**
walkerAmount	16, 32, 64	[30]
TTL	7, 14, 20	preliminary benchmarks

Table 6.8: k-Random Walker parameter values before the Sensitivity Analysis

nodes	packets	*walkerAmount*	TTL
50	100	64	20
50	500		
50	1000		
100	100		
100	500		
100	1000		
200	100		
200	500		
200	1000		

Table 6.9: k-Random Walker routing algorithm Sensitivity Analysis results

The majority of parameter value (ranges) of BeeNet, described in 3.2, are based on the recommendations in [42] . Since the `beeSpawnInterval` is introduced in the routing adaption, it is, in addition to the weight constants α and β, subject of the Sensitivity Analysis. The values before the analysis are shown in 6.10, its results in table 6.11. Interestingly, the optimal combination of α and β varies dependent on the specific networks size, while the optimal `beeSpawnInterval` tends to grow based on the amount of nodes participating in the network.

Parameter	Value (Range)	Source
λ	0.99	[42]
(α, β)	(1,10), (1,1) (10,1)	
beeSpawnInterval	1000 ms, 2000 ms, 3000 ms	preliminary benchmarks

Table 6.10: BeeNet parameter values before the Sensitivity Analysis

nodes	packets	(α, β)	*beeSpawnInterval*
50	100	(1,10)	1000 ms
50	500		
50	1000		
100	100	(1,1)	
100	500		
100	1000		
200	100	(10,1)	3000 ms
200	500		
200	1000		

Table 6.11: BeeNet Sensitivity Analysis results

The parameter value (ranges) of SMNet, described in detail in 3.3, are partly based on [23]. However, since some of the recommended values did not show satisfying results in the preliminary benchmarks, the majority of parameters are subject of the Sensitivity Analysis. Only for the `pseudopodWaitTime` and the `maxPseudopodLimit` parameter the recommendation of [23] is followed for all configurations.
The parameter values before the Sensitivity Analysis are shown in table 6.12, while its results are shown in table 6.13.

Compared to the recommended values in [23], the algorithm shows the best performance for different `minSearchSteps` and `maxPseudopodTTL` values. The optimal `minSearchSteps` is smaller (2 instead of 3) and the optimal `maxPseudopodTTL` is not configuration dependent but constant. Similar to the BeeNet routing algorithm adaption, the optimal `amoebaSpawnInterval` tends to grow proportionally to the P2P networks size. Not surprisingly, due to the existence of the `maxPseudopodLimit`, the algorithm shows to be very robust to changes of the `amountOfSpawnedAmoebas`. Thus, the smallest value of the range is chosen for the competitive analysis.

The `amoebaSpawnInterval` and the `amountOfSpawnedAmoebas` are used in the initialization stage of the algorithm, while the `pseudopodWaitTime`,
the `maxPseudopodLimit`, the `minSearchSteps` and the `maxPseudopodTTL` are used in the vegetative stage of the Dd life-cycle.

Parameter	**Value (Range)**	**Source**
$pseudopodWaitTime$	60000 ms	[23]
$maxPseudopodLimit$	8	
$minSearchSteps$	2, 3, 4	[23], preliminary benchmarks
$amoebaSpawnInterval$	3000 ms, 4000 ms, 5000 ms	preliminary benchmarks
$amountOf SpawnedAmoebas$	3, 4, 5	
$maxPseudopodTTL$	7, 14, 20	

Table 6.12: SMNet parameter values before the Sensitivity Analysis

nodes	packets	*minSearch Steps*	*amoebaSpawn Interval*	*#spawned Amoebas*	*maxPp TTL*
50	100	2	3000 ms	3	7
50	500				
50	1000				
100	100				
100	500				
100	1000				
200	100		5000 ms		
200	500				
200	1000				

Table 6.13: SMNet Sensitivity Analysis results

An additional table that includes descriptions for the parameters of all benchmarked algorithms can be found in the appendix. A.2

6.3 Raw Result Data

The raw result data of the competitive benchmarks are shown in the following tables: AntNet in 6.15, BeeHive in 6.16, Physarum polycephalum in 6.17, Gnutella Flooding in 6.18, k-Random Walker in 6.19, BeeNet in 6.20 and SMNet in 6.21.
The tables contain the average result for each combination of network size levels and data packet levels. In order to display the tables properly, the metrics, selected in 6.1, are abbreviated. The corresponding explanation is provided in table 6.14.

Abbreviation	Explanation
nodes	Amount of P2P nodes participating in the overlay network.
packets	Amount of data packets sent in the benchmark.
delivery ratio	Data packet delivery ratio. Rounded to 2 decimals.
avg. delay	Average data packet delay. Rounded to 2 decimals.
avg. hop count	Average data packet hop count. Rounded to 2 decimals.
routing overhead msg.	Amount of routing overhead messages. Rounded to integers.

Table 6.14: Metric abbreviation explanation for competitive benchmarks

nodes	**packets**	***delivery ratio***	***avg. delay***	***avg. hop count***	***routing overhead msg.***
50	100	1	503.57 ms	4.57	24867
50	500	1	584.42 ms	4.75	24869
50	1000	1	651.21 ms	4.70	24860
100	100	1	749.40 ms	5.95	99008
100	500	1	959.13 ms	6.10	99018
100	1000	1	1179.91 ms	6.00	98507
200	100	1	1382.63 ms	7.65	384921
200	500	1	1933.48 ms	7.55	382837
200	1000	1	2532.09 ms	7.59	376816

Table 6.15: AntNet raw result data

nodes	**packets**	***delivery ratio***	***avg. delay***	***avg. hop count***	***routing overhead msg.***
50	100	1	436.54 ms	4.12	81600
50	500	1	487.46 ms	4.23	81600
50	1000	1	535.80 ms	4.30	81600
100	100	1	524.87 ms	4.96	666400
100	500	1	610.34 ms	5.11	666400
100	1000	1	719.10 ms	4.95	666400
200	100	1	782.78 ms	5.70	3168000
200	500	1	1136.21 ms	5.54	3168000
200	1000	1	1854.37 ms	5.76	3168000

Table 6.16: BeeHive raw result data

nodes	**packets**	***delivery ratio***	***avg. delay***	***avg. hop count***	***routing overhead msg.***
50	100	1	1477.46 ms	12.73	24769
50	500	1	1680.49 ms	12.45	24762
50	1000	1	1967.64 ms	12.57	24741
100	100	1	2434.53 ms	16.54	97143
100	500	1	2923.13 ms	15.73	97143
100	1000	1	3782.02 ms	16.42	96979
200	100	1	4306.41 ms	18.92	368468
200	500	1	6124.41 ms	19.77	368447
200	1000	1	8339.07 ms	20.39	366753

Table 6.17: Physarum polycephalum raw result data

nodes	**packets**	***delivery ratio***	***avg. delay***	***avg. hop count***	***routing overhead msg.***
50	100	1	573.82 ms	2.64	9322
50	500	1	920.18 ms	2.65	46592
50	1000	1	1407.26 ms	2.61	93106
100	100	1	941.84 ms	3.01	19308
100	500	1	1603.11 ms	2.95	96618
100	1000	1	2976.61 ms	3.01	193028
200	100	1	1509.52 ms	3.38	39300
200	500	1	2699.98 ms	3.34	196517
200	1000	1	6141.85 ms	3.36	392984

Table 6.18: Gnutella Flooding raw result data

nodes	**packets**	***delivery ratio***	***avg. delay***	***avg. hop count***	***routing overhead msg.***
50	100	1.00	1256.17 ms	3.70	6300
50	500	1.00	2164.75 ms	3.68	31500
50	1000	1.00	3349.04 ms	3.71	63000
100	100	0.99	2069.93 ms	5.52	6301
100	500	0.99	3991.32 ms	5.30	31504
100	1000	0.99	5828.58 ms	5.50	63008
200	100	0.94	4423.35 ms	7.51	6306
200	500	0.93	7887.77 ms	7.47	31537
200	1000	0.93	10655.75 ms	7.54	63075

Table 6.19: k-Random Walker raw result data

nodes	**packets**	***delivery ratio***	***avg. delay***	***avg. hop count***	***routing overhead msg.***
50	100	1	391.87 ms	3.29	7694
50	500	1	437.62 ms	3.26	7694
50	1000	1	483.96 ms	3.22	7694
100	100	1	577.49 ms	3.84	30136
100	500	1	736.28 ms	3.89	30130
100	1000	1	935.53 ms	3.92	29803
200	100	1	1045.05 ms	5.48	40590
200	500	1	1498.29 ms	5.68	40593
200	1000	1	1805.65 ms	5.52	40582

Table 6.20: BeeNet raw result data

nodes	packets	*delivery ratio*	*avg. delay*	*avg. hop count*	*routing overhead msg.*
50	100	1	418.41 ms	3.89	32004
50	500	1	448.12 ms	3.76	32756
50	1000	1	485.90 ms	3.76	32375
100	100	1	428.65 ms	4.02	116060
100	500	1	500.23 ms	4.13	116943
100	1000	1	622.83 ms	4.07	115895
200	100	1	513.75 ms	4.91	234332
200	500	1	654.04 ms	4.93	233622
200	1000	1	936.32 ms	4.91	233871

Table 6.21: SlimeMoldNet raw result data

6.4 Competitive Analysis

In the following, the result data of the competitive benchmarks is evaluated.

There is only one benchmarked routing algorithm that falls out of line regarding the *data packet delivery ratio* metric. Due to its fully random nature, the k-Random Walker algorithm is not able to deliver all data packets in some instances. The metric shows to get worse for the algorithm, the bigger the network becomes. All other algorithm show a perfect delivery ratio in the benchmarks.

The results for the metric *average data packet delay* are visualized in figures 6.1 and 6.2. At the smallest network size of 50 node, only BeeHive is able to compete with the low delays of the path-based algorithms SMNet and BeeNet. All other algorithms are considerably outperformed by SMNet and BeeNet.

More specifically, for the *average data packet delay* at 50 nodes, SMNet is 6% (100 data packets) and 2% (500 data packets) slower than BeeNet but equally fast at a traffic of 1000 data packets.
SMNet outperforms AntNet by 17% (100 data packets), 23% (500 data packets) and 25% (1000 data packets), BeeHive by 4% (100 data packets), 8% (500 data packets) and 9% (1000 data packets), Physarum polycephalum by 72% (100 data packets), 73% (500 data packets) and 75% (1000 data packets), Gnutella Flooding by 27% (100 data packets),

51% (500 data packets) and 65% (1000 data packets), and k-Random Walker by 67% (100 data packets), 79% (500 data packets) and 85% (1000 data packets).

For the *average data packet delay* at a network size of 50 nodes, BeeNet is 6% (100 data packets) and 2% (500 data packets) faster than SMNet. At 1000 data packets there is practically no difference between BeeNet and SMNet.
BeeNet outperforms AntNet by 22% (100 data packets), 25% (500 data packets) and 26% (1000 data packets), BeeHive by 10% (100 data packets) 10% (500 data packets) and 10% (1000 data packets), Physarum polycephalum by 73% (100 data packets), 74% (500 data packets) and 75% (1000 data packets), Gnutella Flooding by 32% (100 data packets), 52% (500 data packets) and 66% (1000 data packets) and k-Random Walker by 69% (100 data packets), 80% (500 data packets) and 86% (1000 data packets).

At a network size of 100 nodes, SMNet outperforms BeeNet by 26% (100 data packets), 32% (500 data packets) and 33% (1000 data packets), AntNet by 43% (100 data packets), 48% (500 data packets) and 47% (1000 data packets), BeeHive by 18% (100 data packets), 18% (500 data packets) and 13% (1000 data packets), Physarum polycephalum by 82% (100 data packets), 83% (500 data packets) and 84% (1000 data packets), Gnutella Flooding by 54% (100 data packets), 69% (500 data packets) and 79% (1000 data packets), and k-Random Walker by 79% (100 data packets), 87% (500 data packets) and 89% (1000 data packets).

Besides SMNet, as shown above, and BeeHive, BeeNet outperforms all other algorithms regarding the *average data packet delay* at a network size of 100 nodes. BeeNet is 10% (100 data packets), 21% (500 data packets) and 30% (1000 data packets) slower than BeeHive. However, BeeNet outperforms AntNet by 23% (100 data packets), 23% (500 data packets) and 21% (1000 data packets), Physarum polycephalum by 76% (100 data packets), 75% (500 data packets) and 75% (1000 data packets), Gnutella Flooding by 39% (100 data packets), 54% (500 data packets) and 69% (1000 data packets), and k-Random Walker by 72% (100 data packets), 82% (500 data packets) and 84% (1000 data packets).

Especially at the biggest network size of 200 P2P nodes, SMNet outperforms all other algorithms, while BeeNets *average data packet delay* becomes occasionally slightly higher than that of BeeHive. Both, BeeNet and BeeHive have a considerably higher delay than SMNet, while outperforming the rest.
AntNets *average data packet delay* is worse than that of SMNet, BeeNet and BeeHive, in all benchmarked configurations, but is still above average on all levels. SMNet, BeeNet, BeeHive and AntNet show to be very resilient to the network traffic level. Therefore, their *average data packet delay* remains nearly constant when the network traffic load is increased.

More specifically, for the *average data packet delay* at 200 nodes, SMNet dominates and outperforms BeeNet by 51% (100 data packets), 56% (500 data packets) and 48%

(1000 data packets), AntNet by 63% (100 data packets), 66% (500 data packets) and 63% (1000 data packets), BeeHive by 34% (100 data packets), 42% (500 data packets) and 50% (1000 data packets), Physarum polycephalum by 88% (100 data packets), 89% (500 data packets) and 89% (1000 data packets), Gnutella Flooding by 66% (100 data packets), 76% (500 data packets) and 85% (1000 data packets), and k-Random Walker by 88% (100 data packets), 92% (500 data packets) and 91% (1000 data packets).

As shown above, BeeNet is significantly outperformed by SMNet.
Furthermore, BeeNet is 34% (100 data packets) and 32% (500 data packets) slower than BeeHive, but 3% faster at a traffic of 1000 data packets.
Moreover, BeeNet outperforms AntNet by 24% (100 data packets), 23% (500 data packets) and 29% (1000 data packets), Physarum polycephalum by 76% (100 data packets), 76% (500 data packets) and 78% (1000 data packets), Gnutella Flooding by 31% (100 data packets), 45% (500 data packets) and 71% (1000 data packets), and k-Random Walker by 76% (100 data packets), 81% (500 data packets) and 83% (1000 data packets).

Thus, the other 3 algorithms, Gnutella, Physarum polycephalum and k-Random Walker are massively outperformed regarding the *average data packet delay*. Only at a low network traffic load of 100 data packets, Gnutella shows acceptable results. However, the average delay gets worse proportionally to the amount of data packets sent. Physarum polycephalum and the k-Random Walker routing algorithm have the highest *average data packet delay* results. It is about 3 to 4 times higher than that of the other algorithms at a low network traffic load and gets worse rapidly when the load is increased.

The results for the metric *average data packet hop count* are visualized in figures 6.3 and 6.4. Gnutella shows the best results regarding that metric at all network sizes and all traffic levels. If there are multiple duplicates of a data entry, the *average data packet hop count* metric only takes the fastest delivered packet into consideration. Thus, Gnutellas good results are no surprise, since in this routing algorithm, duplicates are flooded to all possible paths and therefore the fastest delivered duplicate of a data packet most likely has taken the path with the least amount of hops.

With the exception of Physarum polycephalum, which is massively outperformed by the other algorithms on all levels, the rest of the routing algorithms show good results. At a network size of 50 nodes, BeeNet, SMNet, AntNet, BeeHive and k-Random Walker are very close.

BeeNet has a slight performance edge, while AntNet is slightly behind of this group performance-wise. As the network size grows to 100, SMNet and BeeNet start getting ahead of AntNet, BeeHive and k-Random Walker. This trend continues at 200 nodes. SMNet takes a clear second place behind Gnutella, closely chased by BeeNet and BeeHive. Gnutella still dominates, AntNet and the k-Random Walker algorithm fall behind by a considerable amount.

The results for the metric *routing overhead messages* are visualized in figures 6.5 and 6.6. While showing good results for the other metrics, BeeHives performance regarding the routing overhead is abysmal on all levels. Especially when the network size grows, the performance get exponentially worse. This is no surprise, since BeeHive implements constantly periodically flooding of routing information of all network nodes.

On the other side of the spectrum is BeeNet, which overall shows the lowest amount of routing overhead messages on all network sizes (50, 100, 200). Moreover, the algorithm shows to be resilient to an increment of the data packet network traffic.
BeeNet is only outperformed in regard to the amount of *routing overhead messages* at low levels of network traffic. This is where Gnutella Flooding performs very well. However, as the network size and the data traffic grows, Gnutella gets outperformed quickly.

The k-Random Walker routing algorithm produces a exceptionally low amount of routing overhead messages. It only grows with the amount of data packets sent and is independent of the network sizes. However, the major drawback of the algorithm is the bad *data packet delivery ratio* described above. This relativizes the good performance of k-Random Walker in this category considerably.

Due to a similar agent based approach and the same spawn interval of agents, AntNet and Physarum polycephalum have practically the same amount of routing overhead messages on all levels. The main reason these two algorithms have a bigger routing overhead than BeeNet is their agent spawn interval. This is where the path-based approach of BeeNet shines. It still performs very well with a rather low spawn interval of bee agents compared to AntNet and Physarum polycephalum.

Due to the process of iterative spawning of pseudopods in the vegetative phase of the algorithm and its multi-phase nature, the SMNet algorithm shows beyond average results at the smallest network size of 50 nodes. However, SMNets amount of *routing overhead messages* gets more and more competitive as the network size and the network traffic grows. Moreover, SMNet shows to be very resilient to an increment of the network traffic load.
While still performing beyond average at the network size of 100 P2P nodes, at 200 P2P nodes and a traffic amount of 500 data packets, SMNet starts outperforming AntNet, Physarum polycephalum and Gnutella. Only BeeNet and the k-Random Walker algorithm remain with better performance. However, again, it is important to note the bad *data packet delivery ratio* of the k-Random Walker algorithm, especially at a network size of 200 nodes. This puts the perceived good performance of the k-Random Walker algorithm into perspective.

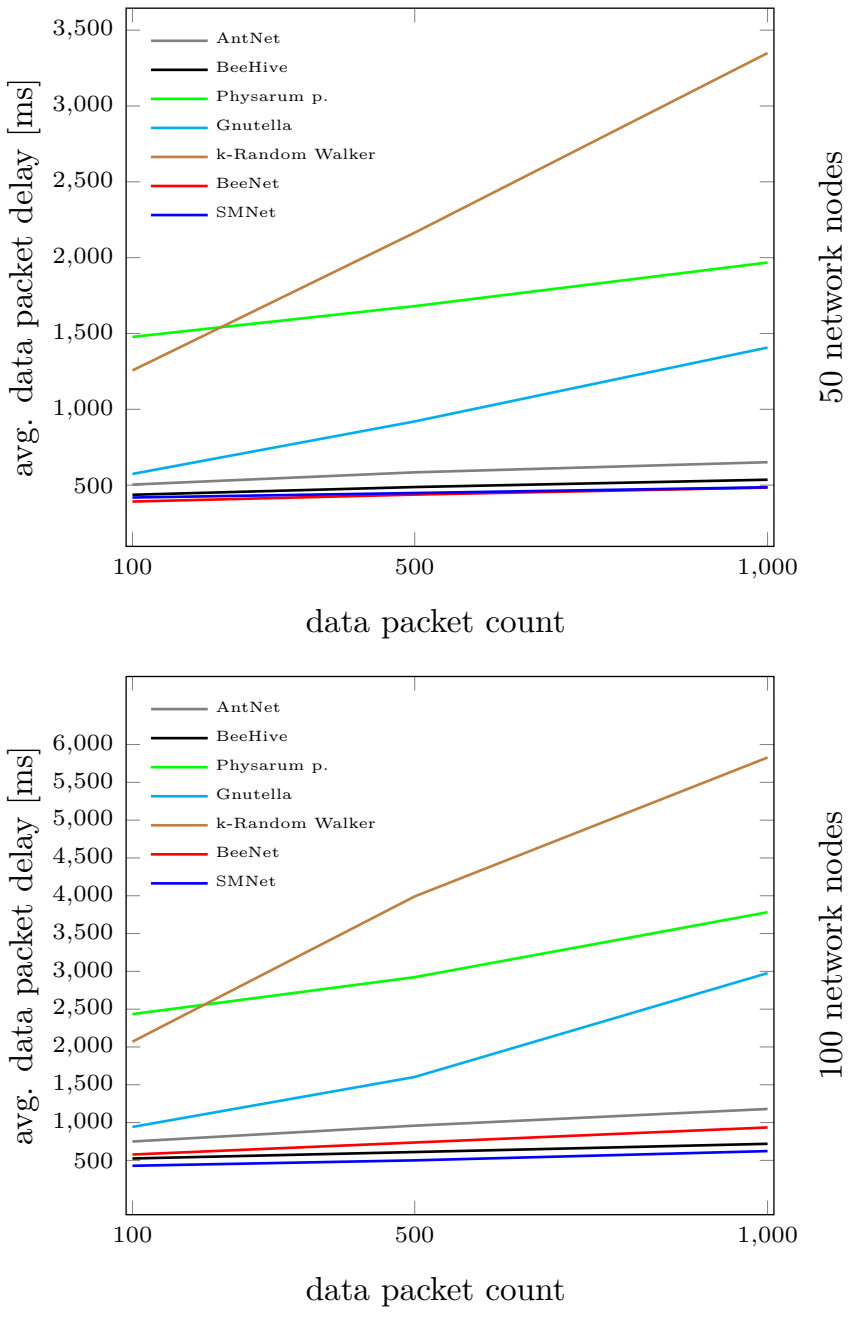

Figure 6.1: Average data packet delay results for network sizes 50, 100

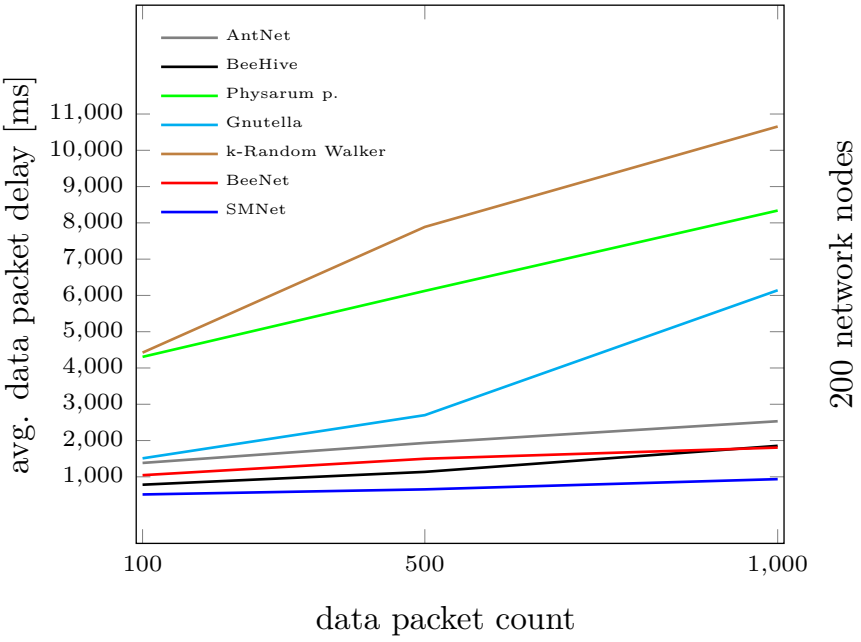

Figure 6.2: Average data packet delay results for network size 200

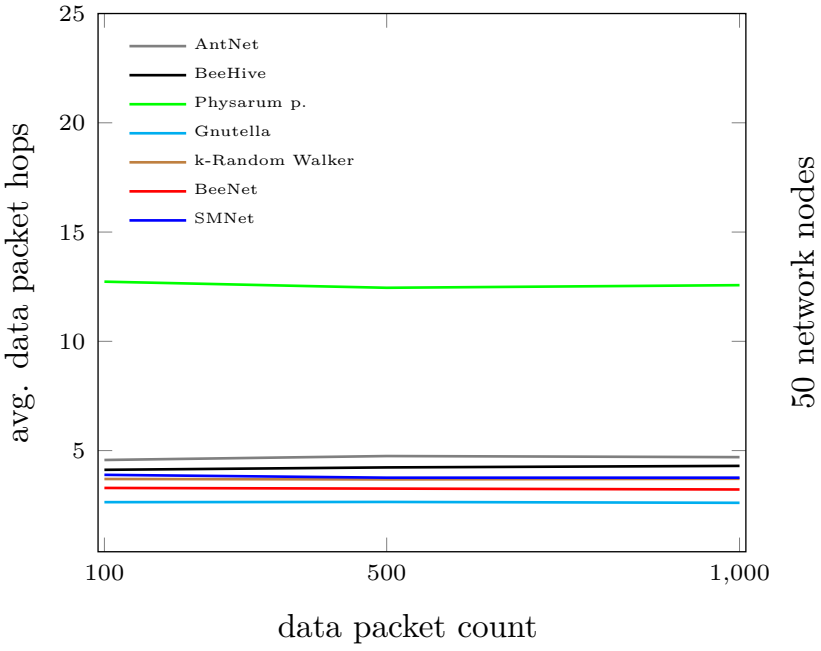

Figure 6.3: Average data packet hop count results for network size 50

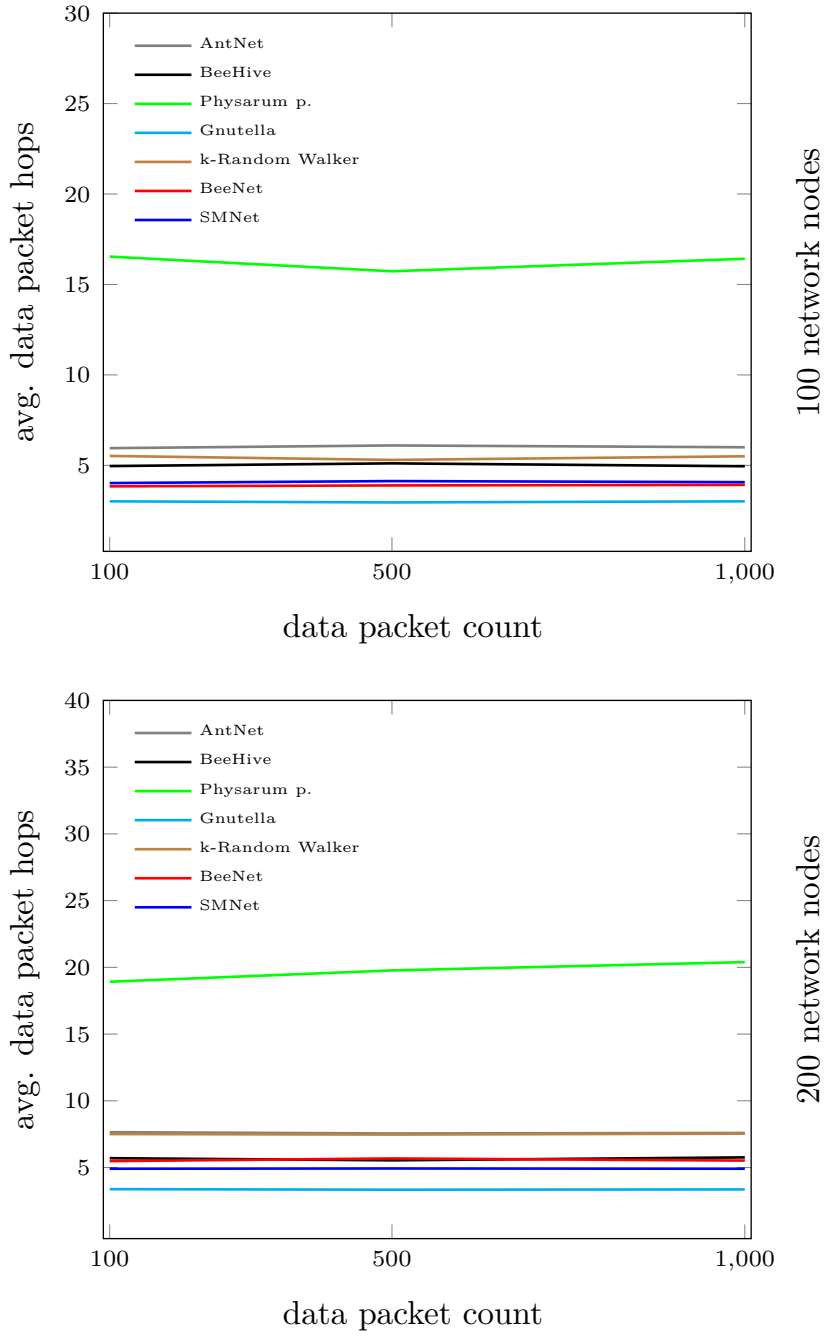

Figure 6.4: Average data packet hop count results for network sizes 100, 200

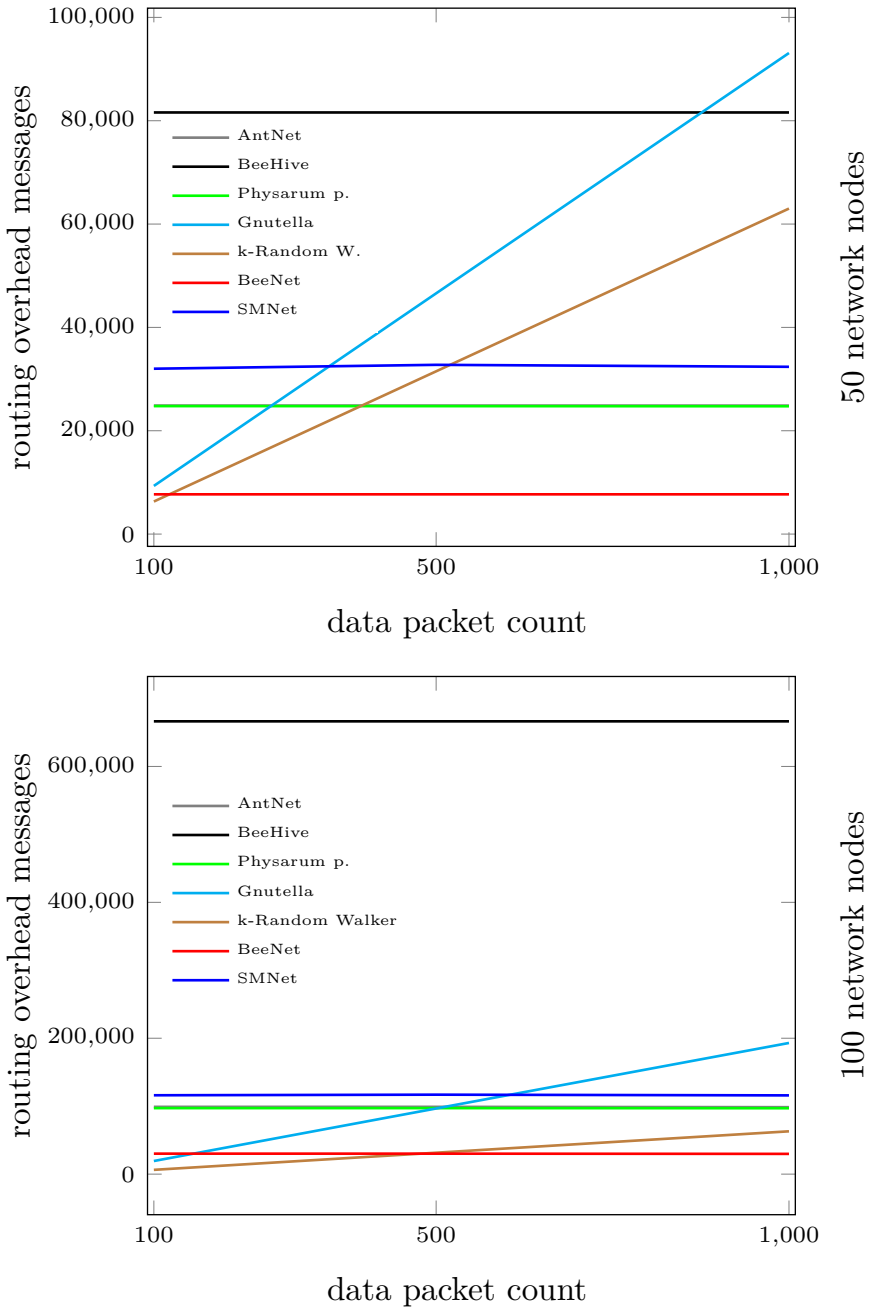

Figure 6.5: Amount of routing overhead messages for network sizes 50, 100

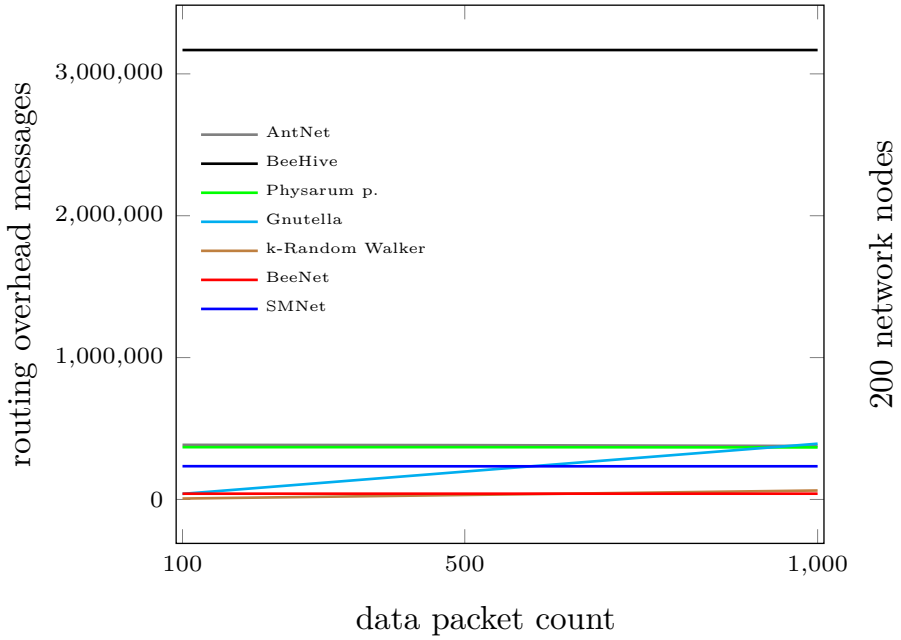

Figure 6.6: Amount of routing overhead messages for network size 200

6.5 Statistical Analysis

For an in-depth evaluation and comparison of SMNet and BeeNet to the other benchmarked routing algorithm, statistical analysis, described in [40], is done.

More specifically, one-way ANOVA tests are performed with the following setup [40]: H_0 is the null-hypothesis stating that there is no significant difference between the metric value M of 2 algorithm A and B and H_1 is the corresponding alternative-hypothesis. Thus, if H_0 is rejected, not enough data is available to determine if A or B is significantly better regarding metric M. Moreover, if H_0 is rejected, H_1 is concluded [40].

The tests are carried out for SMNet and BeeNet in separate tests, for all combinations of network sizes level (L_n, M_n, H_n) and data entry amount levels (L_d, M_d, H_d) described in section 6.1. As metric M, the *average data packet delay* is chosen, since it represents the performance of an algorithm really well and is of great importance, especially in time critical scenarios. The significance level for the ANOVA tests is chosen as $\alpha = 0,05$. [40]

First, SMNet takes the role of algorithm A in the test and all other benchmarked algorithms (including BeeNet) take the role of algorithm B. The results of this tests are shown in tables 6.22 and 6.23.
Then, BeeNet takes the role of algorithm A and all other benchmarked algorithms

(including SMNet) take the role of algorithm B. The results of this tests are shown in tables 6.24 and 6.25.

For all result tables of the statistical tests, if H_0 is concluded, the column h in the table has the value 0. If H_0 is rejected, and therefore there is a significant difference between the tested algorithms, h has the value 1 if algorithm A performs better than algorithm B and -1 otherwise.

Example 6.5.1 *For a network size of 100 P2P network nodes and 500 data packets sent, SMNets mean average data packet delay is 500,23 and the standard deviation is 15,93. For the same configuration, AntNets mean average data packet delay is 959.14 and the standard deviation is 41,10.*
The result of the one-way ANOVA test is $p = 7,06931\ 10^{-10}$.
Since the significance level is chosen $\alpha = 0.05$ *and* $\alpha > p$, *the null-hypothesis* H_0 *is rejected and the alternative-hypothesis* H_1 *concluded.*
The test proves the significant difference of the average data packet delay metric values. Thus, it can be concluded that SMNet performs significantly better at this configuration than AntNet. Therefore the h value is set to 1 in the result table.

As tables 6.22 and 6.23 show, SMNet performs significantly better than all other algorithms at all the network size levels of 50, 100 and 200 nodes with very few exceptions.
Only at a network size of 50 and a network traffic load of 100, BeeNet performs better and H_0 can not be rejected when compared to BeeHive.
Furthermore, when comparing SMNet to BeeNet on the data packet traffic levels 500 and 1000 at the same network size of 50 nodes, there is not enough data to conclude which routing algorithm performs better. Thus, H_0 can not be rejected.

As shown in tables 6.24 and 6.25, BeeNet shows good performance in the statistical analysis. At a network size of 50 nodes and a traffic level of 100 data packets, it performs significantly better than all other benchmarked algorithms.
For a higher data packet count 500 and 1000 at the same network size of 50 nodes, it also shows very competitive performance. Only when compared to SMNet in those configurations, H_0 can not be rejected. All other routing algorithms are outperformed at these configurations.
For all other configurations at the network size of 100 and 200 nodes, BeeNet takes the third place after SMNet and BeeHive. However, the results at the network size of 200 and 1000 data packets are slightly different. BeeNets performance level increases. While, H_0 can not be rejected when compared to BeeHive, BeeNet is still outperformed significantly by SMNet.

	mean ± stdev	p-value	h
	50 nodes 100 data packets		
AntNet	503.57 ± 22.79	2.7329E-08	1
BeeHive	436.54 ± 37.19	0.18180051	0
Physarum p.	1477.46 ± 100.34	1.602E-17	1
Gnutella	573.82 ± 39.43	1.2336E-09	1
k-Random Walker	1256.17 ± 112.64	7.1709E-15	1
BeeNet	391.87 ± 25.97	0.01590813	-1
SMNet	418.41 ± 17.91	-	-
	50 nodes 500 data packets		
AntNet	584.42 ± 18.37	5.3072E-12	1
BeeHive	487.46 ± 32.39	0.00433343	1
Physarum p.	1680.49 ± 75.27	9.0244E-21	1
Gnutella	920.18 ± 45.57	8.1788E-17	1
k-Random Walker	2164.75 ± 124.54	1.3237E-19	1
BeeNet	437.62 ± 14.64	0.19880675	0
SMNet	448.12 ± 20.13	-	-
	50 nodes 1000 data packets		
AntNet	651.21 ± 37.04	7.0691E-10	1
BeeHive	535.8 ± 27.79	0.00048294	1
Physarum p.	1967.64 ± 160.23	1.5461E-16	1
Gnutella	1407.26 ± 66.51	3.0311E-19	1
k-Random Walker	3349.04 ± 156.51	8.3102E-22	1
BeeNet	483.96 ± 16.28	0.83802976	0
SMNet	485.90 ± 24.64	-	-
	100 nodes 100 data packets		
AntNet	749.40 ± 46.27	9.7149E-14	1
BeeHive	524.87 ± 29.88	1.3151E-07	1
Physarum p.	2434.53 ± 194.32	1.9911E-17	1
Gnutella	941.84 ± 45.30	1.8915E-17	1
k-Random Walker	2069.93 ± 205.93	1.881E-15	1
BeeNet	577.49 ± 39.31	3.7162E-09	1
SMNet	428.65 ± 20.82	-	-
	100 nodes 500 data packets		
AntNet	959.13 ± 41.10	1.5381E-17	1
BeeHive	610.34 ± 32.56	1.6491E-08	1
Physarum p.	2923.13 ± 178.95	1.5533E-19	1
Gnutella	1603.11 ± 114.70	7.4834E-17	1
k-Random Walker	3991.32 ± 217.07	7.0192E-21	1
BeeNet	736.28 ± 56.30	1.874E-10	1
SMNet	500.23 ± 15.93	-	-

Table 6.22: SMNet ANOVA results. (part 1)

	mean ± stdev	p-value	h
	100 nodes 1000 data packets		
AntNet	1179.91 ± 44.92	2.9732E-18	1
BeeHive	719.10 ± 23.12	6.7697E-09	1
Physarum p.	3782.02 ± 276.97	3.1936E-18	1
Gnutella	2976.61 ± 151.90	1.4931E-20	1
k-Random Walker	5828.58 ± 138.53	1.925E-27	1
BeeNet	935.53 ± 56.02	2.0693E-12	1
SMNet	622.83 ± 18.96	-	-
	200 nodes 100 data packets		
AntNet	1382.63 ± 39.49	1.0406E-20	1
BeeHive	782.78 ± 119.27	2.3613E-06	1
Physarum p.	4306.41 ± 417.27	1.8414E-16	1
Gnutella	1509.52 ± 242.30	1.7024E-10	1
k-Random Walker	4423.35 ± 613.17	8.6684E-14	1
BeeNet	1045.05 ± 53.16	1.3776E-15	1
SMNet	513.75 ± 38.82	-	-
	200 nodes 500 data packets		
AntNet	1933.48 ± 61.24	2.2994E-22	1
BeeHive	1136.21 ± 117.96	2.1212E-10	1
Physarum p.	6124.41 ± 293.01	4.9293E-22	1
Gnutella	2699.98 ± 227.10	2.2067E-16	1
k-Random Walker	7887.77 ± 406.70	1.1403E-21	1
BeeNet	1498.29 ± 91.85	2.546E-16	1
SMNet	654.04 ± 24.33	-	-
	200 nodes 1000 data packets		
AntNet	2532.09 ± 157.21	4.0924E-17	1
BeeHive	1854.37 ± 156.24	5.7019E-13	1
Physarum p.	8339.07 ± 402.21	6.5036E-22	1
Gnutella	6141.85 ± 464.06	4.3835E-18	1
k-Random Walker	10655.75 ± 640.00	1.9268E-20	1
BeeNet	1805.65 ± 163.03	2.8591E-12	1
SMNet	936.32 ± 38.80	-	-

Table 6.23: SMNet ANOVA results. (part 2)

	mean ± stdev	p-value	h
	50 nodes 100 data packets		
AntNet	503.57 ± 22.79	6.34115E-09	1
BeeHive	436.54 ± 37.19	0.00598568	1
Physarum p.	1477.46 ± 100.34	1.3898E-17	1
Gnutella	573.82 ± 39.43	3.9322E-10	1
k-Random Walker	1256.17 ± 112.64	5.2571E-15	1
BeeNet	391.87 ± 25.97	-	-
SMNet	418.41 ± 17.91	0.01590813	1
	50 nodes 500 data packets		
AntNet	584.42 ± 18.37	1.1857E-13	1
BeeHive	487.46 ± 32.39	0.00032073	1
Physarum p.	1680.49 ± 75.27	5.827E-21	1
Gnutella	920.18 ± 45.57	2.7305E-17	1
k-Random Walker	2164.75 ± 124.54	1.0661E-19	1
BeeNet	437.62 ± 14.64	-	-
SMNet	448.12 ± 20.13	0.19880675	0
	50 nodes 1000 data packets		
AntNet	651.21 ± 37.04	1.2586E-10	1
BeeHive	535.8 ± 27.79	7.65E-05	1
Physarum p.	1967.64 ± 160.23	1.346E-16	1
Gnutella	1407.26 ± 66.51	1.559E-19	1
k-Random Walker	3349.04 ± 156.51	7.2612E-22	1
BeeNet	483.96 ± 16.28	-	-
SMNet	485.90 ± 24.64	0.83802976	0
	100 nodes 100 data packets		
AntNet	749.40 ± 46.27	4.7568E-08	1
BeeHive	524.87 ± 29.88	0.00340935	-1
Physarum p.	2434.53 ± 194.32	1.0034E-16	1
Gnutella	941.84 ± 45.30	1.9288E-13	1
k-Random Walker	2069.93 ± 205.93	1.2385E-14	1
BeeNet	577.49 ± 39.31	-	-
SMNet	428.65 ± 20.82	3.7162E-09	-1
	100 nodes 500 data packets		
AntNet	959.13 ± 41.10	7.545E-09	1
BeeHive	610.34 ± 32.56	8.7655E-06	-1
Physarum p.	2923.13 ± 178.95	2.0811E-18	1
Gnutella	1603.11 ± 114.70	2.8599E-14	1
k-Random Walker	3991.32 ± 217.07	4.1817E-20	1
BeeNet	736.28 ± 56.30	-	-
SMNet	500.23 ± 15.93	1.874E-10	-1

Table 6.24: BeeNet ANOVA results. (part 1)

	mean ± stdev	p-value	h
	100 nodes 1000 data packets		
AntNet	1179.91 ± 44.92	2.8509E-09	1
BeeHive	719.10 ± 23.12	1.3323E-09	-1
Physarum p.	3782.02 ± 276.97	2.7764E-17	1
Gnutella	2976.61 ± 151.90	5.1617E-19	1
k-Random Walker	5828.58 ± 138.53	1.9354E-26	1
BeeNet	935.53 ± 56.02	-	-
SMNet	622.83 ± 18.96	2.0693E-12	-1
	200 nodes 100 data packets		
AntNet	1382.63 ± 39.49	3.8374E-12	1
BeeHive	782.78 ± 119.27	5.5302E-06	-1
Physarum p.	4306.41 ± 417.27	2.7884E-15	1
Gnutella	1509.52 ± 242.30	1.3256E-05	1
k-Random Walker	4423.35 ± 613.17	1.095E-12	1
BeeNet	1045.05 ± 53.16	-	-
SMNet	513.75 ± 38.82	1.3776E-15	-1
	200 nodes 500 data packets		
AntNet	1933.48 ± 61.24	2.7263E-10	1
BeeHive	1136.21 ± 117.96	4.5306E-07	-1
Physarum p.	6124.41 ± 293.01	2.1509E-20	1
Gnutella	2699.98 ± 227.10	7.3384E-12	1
k-Random Walker	7887.77 ± 406.70	1.5864E-20	1
BeeNet	1498.29 ± 91.85	-	-
SMNet	654.04 ± 24.33	2.546E-16	-1
	200 nodes 1000 data packets		
AntNet	2532.09 ± 157.21	7.1769E-09	1
BeeHive	1854.37 ± 156.24	0.50372669	0
Physarum p.	8339.07 ± 402.21	2.1806E-20	1
Gnutella	6141.85 ± 464.06	2.924E-16	1
k-Random Walker	10655.75 ± 640.00	1.7405E-19	1
BeeNet	1805.65 ± 163.03	-	-
SMNet	936.32 ± 38.80	2.8591E-12	-1

Table 6.25: BeeNet ANOVA results. (part 2)

6.6 Scalability Analysis

It is from substantial importance for an algorithm, used in P2P network environments, to be resistant to an increase of the traffic load and the network size, since it has to be ensured that all nodes of a P2P networks are able to keep operating normally in that scenario [23].
In order to evaluate how the benchmarks react to an increment of the network size and the data packet traffic, a scalability analysis, based on [22], is carried out.

To enable the evaluation of systems regarding their scalability, a scalability metric is proposed [22]:

$$\psi = \frac{F(\lambda_2, QoS, C_2)}{F(\lambda_1, QoS, C_1)} \tag{6.1}$$

where F is a performance evaluating function that is dependent on λ, which evaluates the rate of providing services to users, a set of quality of service parameters (QoS), and a cost parameter C, that specifies the cost of providing the service to users [22].

For the scalability analysis, the same adaption of this metric, provided by [23], is used. To evaluate an algorithms performance regarding its scalability, the following function P is used [23]:

$$P(L, R) = \frac{1}{M}, \quad M \neq 0 \tag{6.2}$$

where M are the average messages per node, L is the load and R the resources. In the scenario of routing, L is the amount of *routing overhead messages* and R the amount of P2P nodes participating in the network. Therefore, M is calculated by:

$$M = \frac{L}{R} = \frac{routing\ overhead\ messages}{network\ size} \tag{6.3}$$

Using the performance function P, the scalability metric for the specific scenario, called *load scalability*, is defined [23]:

$$\psi(k) = \frac{P(kL, kR)}{P(L, R)}, \quad k > 0 \tag{6.4}$$

where the scale factor k is an integer. When $\psi \geq 1$ the routing algorithm scales well. Otherwise, the algorithm is considered not to scale. [23]

The scalability of AntNet, BeeHive, Physarum polycephalum, Gnutella Flooding, k-Random Walker, BeeNet and SMNet is evaluated in the following. The scalability is analyzed at the network size levels of 50, 100, 200 nodes for the scaling factor $k = 2$ and 50, 200 for $k = 4$. For $k = 2$, the initial load is chosen as 250 and 500 data packets, where as for $k = 4$ it is chosen as 250 data packets.

Example 6.6.1 *For a network size of 50 P2P network nodes and 250 data packets sent, BeeNets amount of routing overhead messages is 7693 and therefore* $M_{50,250} = 153{,}88$.

For a network size of 100 P2P network nodes and 500 data packets sent, BeeNets amount of routing overhead messages is 30130 and therefore $M_{100,500} = 301,3$.
The load scalability is calculated by

$$\psi = \frac{1/M_{100,500}}{1/M_{50,250}} = 0.51 \tag{6.5}$$

In order to enable the analysis, additional benchmarks for the network size of 50 nodes and a traffic level of 250 data packets were executed. Parameters chosen for all the algorithms are the same as for 50 nodes and 100 data packets described in the sensitivity analysis 6.2. The additional benchmark results are shown in A.1.
For all described configurations, M is calculated. The values are stated in table 6.26.

The results of the scalability analysis for $\psi(2)$ are shown in table 6.27 and for $\psi(4)$ in table 6.28.

	(nodes / data packets)		
Algorithms	**(50 / 250)**	**(100 / 500)**	**(200 / 1000)**
AntNet	497.18	990.18	1884.08
BeeHive	1632	6664	15840
Physarum p.	495.36	971.43	1833.77
Gnutella	465.36	966.18	1964.92
k-Random Walker	315	315.04	315.38
BeeNet	153.88	301.3	202.91
SMNet	643.76	1169.43	1169.36

Table 6.26: Average routing overhead messages M per (node/data packet) level

	Initial Load	
Algorithms	**50**	**100**
AntNet	0.5	0.53
BeeHive	0.24	0.42
Physarum p.	0.51	0.53
Gnutella	0.48	0.49
k-Random Walker	1	1
BeeNet	0.51	1.48
SMNet	0.55	1

Table 6.27: Results of the Scalability Analysis with k=2

	Initial Load
Algorithms	**50**
AntNet	0.26
BeeHive	0.1
Physarum p.	0.27
Gnutella	0.24
k-Random Walker	1
BeeNet	0.76
SMNet	0.55

Table 6.28: Results of the Scalability Analysis with k=4

Interestingly, for SMNet, the load scalability ψ improves significantly as the initial load is increased. For k=2 and an initial load of 50, the scalability is average with $\psi(2) = 0.55$. However, for an initial load of 100, the load scalability reaches a good load scalability value of $\psi(2) = 1$.
For the same scale factor, BeeNet shows similar results for an initial load of 50, $\psi(2) = 0.51$, and improves even more for an initial load of 100, $\psi(2) = 1.48$.
Thus, both of the routing algorithm proposed in this thesis show very good scalability as the initial load grows.

The k-Random Walker routing algorithm shows a load scalability of $\psi(2) = 1$. Both, SMNet (0.51) and BeeNet (0.55) show only about half the load scalability for k=2. However, as the initial load grows, SMNet reaches the same load scalability of 1 as k-Random Walker. BeeNet outperforms it considerably with a value of 1.48.

AntNet, with $\psi(2) = 0.5$, Physarum polycephalum, with $\psi(2) = 0.51$, and Gnutella Flooding, with $\psi(2) = 0.49$, show similar results as BeeNet (0.51) and SMNet (0.55) at a low initial load and k=2. However, at an initial load of 100, they all fall behind significantly. BeeHive can't compete regarding the scalability on any initial load level.

At a scale factor k = 4 and an initial load of 50, the k-Random Walker algorithm shows, with a load scalability value of $\psi(4) = 1$, better results than all other routing algorithms. This stems from the fact that the algorithm always sends the same amount of walkers per data packet, independently of the network sizes. As discussed in the competitive analysis 6.4, while this behavior results in good scalability, the downside is a decreasing *data packet delivery ratio* as the network grows.

Besides the k-Random walker algorithm, BeeNet, with a load scalability of $\psi(4) = 0.76$ and SMNet, with a load scalability of $\psi(4) = 0.55$ outperform all other evaluated routing algorithms. AntNet, with $\psi(4) = 0.26$, Physarum polycephalum, with $\psi(4) = 0.27$, and Gnutella Flooding, with $\psi(4) = 0.24$ again show similar results.

BeeHive also shows the worst scalability of all benchmark routing algorithms on this level with $\psi(4) = 0.1$.

While BeeNet and SMNet are outperformed by k-Random Walker algorithm at an initial load of 50 and a scale factor of 2, they do not have the same major drawbacks as discussed above.
Furthermore, for BeeNet and SMNet, a change of the bee or amoeba spawn interval influences the scalability of the algorithms. Moreover, the amount of spawned pseudopods by amoebas and the conditions for amoebas to transition to the aggregative state are further considerable factors for SMNet that influence the routing algorithms scalability.

6.7 Summary

In this chapter, BeeNet and SMNet, the routing algorithms proposed in this thesis, are evaluated and compared to AntNet, BeeHive, the Physarum polycephalum routing algorithm, Gnutella Flooding and k-Random Walker.

The benchmark methodology is described at the beginning of the chapter. After an extensive sensitivity analysis in which the optimal parameters for all benchmark configurations are determined, the competitive benchmarks are carried out.

Especially for the *average data packet delay* and the *average hop count* metric, the results show only very few occasions where all other algorithm are not outperformed by SMNet and BeeNet. Especially SMNet shows very good results regarding the *average data packet delay* metric.
This is proven statistically for the *average data packet delay* by one-way ANOVA tests in the statistical analysis.

Lastly, a scalability analysis is done for all seven routing algorithms. Besides the k-Random Walker routing algorithm which has to deal with major drawbacks regarding the *delivery ratio* of data packets for its good scalability, BeeNet and SMNet outperform all other evaluated routing algorithms. Especially BeeNet shows very good scalability as the amount of participating P2P network nodes and the data packet traffic level grows.

CHAPTER 7

Future Work & Conclusion

In this chapter possible future improvements to the framework and the adapted routing algorithms BeeNet and SMNet are discussed. After that, a final conclusion to the master thesis is drawn.

7.1 Future Work

After this master thesis, there still are some possible topics for future research work:

- **Transmission interference:** Although the framework provides an interface in the I/O Peers Service `sendEntriesService`, which allows to implement additional transmission interference such as applying a drop rate to traffic, there is still a lot of room for improvement. The implementation of churn simulation would be the next logical step, since it is characteristic for the dynamic nature of P2P overlay networks. This should technically be no problem: Since the implementation of the framework, the Peer Model Java implementation [8] progressed in its development and now supports the dynamic joining and leaving of Peer Spaces by peers. With the support of churn a very interesting research field becomes accessible. Not only the behavior of routing algorithms when facing churn should be researched, but also different churn models like [51], [15] and [47].
- **Limitations:** As described in section 5.6, some limitations exist in the proposed framework. When the Peer Model Java implementation [8] progresses further in development and provides features like the determination of entry amounts of specific types in peer containers, the framework may be extended to support the determination of queue sizes, which is an important decision factor for some routing algorithms.
- **Usability & Output Visualization:** There is also room for advancement regarding the frameworks usability. The biggest upgrade would be the development of a

graphical user interface which would improve the user experience when configurating and executing benchmarks in the framework. Furthermore, more default output options of benchmark results, like automatically created plots, would shorten the amount of time needed to visualize the output of benchmarks. The application is well prepared for diverse output formats by providing a generic interface in the Statistic Peers `StatisticsOutputService`.

The adapted routing algorithms BeeNet and SMNet also offer an interesting field of future work. First, possible algorithm specific improvements are listed. Then improvements and research topics which are interesting for both, BeeNet and SMNet, are discussed.
An interesting aspect for which research could lead to an improvement of SMNet is:

- **Vegetative Movement:** In this phase amoebas experience new paths. Like in the adapted algorithm [40], pseudopods choose their next hop randomly. It would be very interesting to analyze SMNets behavior and performance when a more sophisticated transition method for pseudopods is used.

BeeNet offers similar research possibilities:

- **Bee Transition:** Although the currently used transition functions show good results, it would be interesting to develop different approaches for forager and follower movement behavior to evaluate and compare them to the existing solution. This might makes the routing algorithm even more efficient.

For both adapted algorithms, possible areas of improvements and research are:

- **Transmission interference:** Since both algorithms have only been benchmarked in a static P2P network environment, it would be very interesting to analyze the adaptiveness of SMNet and BeeNet to additional transmission inteference (e.g. dropping of packets) and churn. There might be room for optimization for both algorithms.

- **Memory usage:** Both algorithms use experienced paths to route data packets to a destination. This might becomes inefficient as the network grows to very high number of nodes, especially on low-memory embedded devices. An interesting field of research is therefore the optimization of the algorithms memory usage. There might be more efficient ways to store routing relevant information experienced by the software agents.

- **Large Scale Tests:** The benchmark results show that BeeNet and SMNet perform well when the amount of network nodes and the data packet traffic level increases. It would be interesting to research, if this trend continues as the network sizes and the amount of sent data packets is increased further. Therefore, benchmarks in larger environments should be carried out.

7.2 Conclusion

There are two goals achieved by this master thesis.

The first goal of the thesis is to create a Peer Model based benchmarking framework, that enables the fair and systematic benchmarking and comparison of routing algorithms in unstructured P2P networks. The framework provides a specific pattern for the implementation of routing algorithms and benchmarks them in their abstracted form, independently of a specific P2P application protocol or use case.

The benchmarking framework shows to fulfill all defined requirements. It provides a clear pattern for the structuring of routing algorithms that enables their fair comparison. Based on this pattern, the frameworks component-based architecture is built. This clearly structured architecture is enabled by the use of the Peer Model and allows new users to get a grip of the basic concepts very quickly. Moreover, the framework offers extensive configurability with provided default values that fit a lot of scenarios. Algorithms are implemented using generic interfaces that serve as additional guideline for locating specific algorithm functionality and which enable easy exchangeability of algorithms and algorithm components.

The second goal is to use the created framework to adapt two existing swarm-intelligent algorithms to the domain of routing from a different domain. The adapted algorithms are Bee Algorithm [42], based on the foraging behavior of honey bees and used for distributed load balancing, and SMP2P [40], which makes use of the Dictyostelium discoideum slime molds life-cylcle to search for resources in P2P networks. Both algorithm make use path-based information that is provided by intelligent software agents which experience these paths when they travel the network. The resulting algorithms BeeNet and SMNet are implemented, evaluated and compared to a diverse range of existing routing algorithms using the benchmarking framework.

SMNet shows to outperform all other benchmarked routing algorithm regarding the average delivery delay of data packets with growing amount of network nodes and data packet traffic. BeeNet takes the overall second place right after SMNet. Regarding algorithm scalability, SMNet and BeeNet show good results for a growing network size and provide considerable resilience to high loads of traffic. They are only outperformed by k-Random Walker in small networks and low traffic loads. However, BeeNet and SMNet show, in contrast to k-Random Walker, a perfect delivery ratio in all benchmarks.

The thesis shows that it is possible to provide a framework, based on a generic pattern, for routing in unstructured P2P networks which enables the fair and systematic evaluation and comparison of routing algorithms in unstructured P2P networks. Additionally, two new routing algorithms are provided that show to perform very well in high traffic networks.

APPENDIX A

Appendix

A.1 Additional Benchmarks for the Scalability Analysis

algorithms	***delivery ratio***	***avg. delay***	***avg. hop count***	***routing overhead msg.***
AntNet	1	541.61 ms	4.67	24859
BeeHive	1	493.59 ms	4.41	81600
Physarum p.	1	1522.42 ms	12.27	24768
Gnutella	1	739.23 ms	2.62	23269
k-Random Walker	1	1815.66 ms	3.60	15750
BeeNet	1	390.17 ms	3.23	7694
SlimeMoldNet	1	429.69 ms	3.76	32188

Table A.1: Benchmark results for 50 nodes and 250 data packets

Parameter	Description
BeeNet	
$beeSpawnInterval$	After this time interval has passed, a new bee is spawned
(α, β)	Weights arc fitness and heuristic distance in transition rule
λ	Probability of follower bee to follow the preferred path
SlimeMoldNet	
$amoebaSpawnInterval$	After this time interval has passed, a new amoeba is spawned
$pseudopodWaitTime$	Maximum wait time for pseudopods to return in a vegetative step
$amountOf$ $SpawnedAmoebas$	Amount of amoebas spawned at a node after $amoebaSpawnInterval$ has passed
$maxPseudopodLimit$	Amount of pseudopods spawned at a vegetative step
$maxPseudopodTTL$	Max. amount of hops before pseudopod termination
$minSearchSteps$	Min. amount of veg. steps before aggregation allowed
AntNet	
$antSpawnInterval$	After this time interval has passed, a new ant is spawned
TTL	Maximum travel time of ant before termination
α	Weights output queue size for ant next hop selection
η	Weights newly calculated mean and variance
c	Constant for observation window calculation
c_1	Constant for calculation of reinforcement value
c_2	Constant for calculation of reinforcement value
γ	Confidence interval for calculation of reinforcement value
a	Constant for squashing the reinforcement value
BeeHive	
$beeSpawnInterval$	After this time interval has passed, a new bee is spawned
$initializationInterval$	Time interval in which foraging regions are built
$shortDistanceLimit$	Hop limit of short distance bees
$longDistanceLimit$	Hop limit of long distance bees
Physarum polycephalum	
$agentSpawnInterval$	After this time interval has passed, a new agent is spawned
ϵ	Weights newly experienced path
Gnutella Flooding	
TTL	Hop limit for data packets
k-Random Walker	
$walkerAmount$	Amount of random walkers per sent data packet
TTL	Maximum hop limit of walkers

Table A.2: Parameter description of benchmarked algorithms

Bibliography

[1] Matteo Agosti et al. “P2pam: a framework for peer-to-peer architectural modeling based on peersim”. In: *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). 2008, p. 22.

[2] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. “A survey of peer-to-peer content distribution technologies”. In: *ACM Computing Surveys (CSUR)* 36.4 (2004), pp. 335–371.

[3] Albert-László Barabási and Réka Albert. “Emergence of scaling in random networks”. In: *science* 286.5439 (1999), pp. 509–512.

[4] John Buford, Heather Yu, and Eng Keong Lua. *P2P networking and applications*. Morgan Kaufmann, 2009.

[5] John F Buford and Heather Yu. “Peer-to-peer networking and applications: Synopsis and research directions”. In: *Handbook of Peer-to-Peer Networking*. Springer, 2010, pp. 3–45.

[6] Scott Camazine and James Sneyd. “A model of collective nectar source selection by honey bees: self-organization through simple rules”. In: *Journal of theoretical Biology* 149.4 (1991), pp. 547–571.

[7] Matteo Casadei et al. “A self-organizing approach to tuple distribution in large-scale tuple-space systems”. In: *International Workshop on Self-Organizing Systems*. Springer. 2007, pp. 146–160.

[8] Stephan Cejka. “Enabling Scalable Collaboration by Introducing Platform-Independent Communication for the Peer Model”. MA thesis. Vienna University of Technology, unpublished.

[9] Stefan Craß, Gerson Joskowicz, Martin Novak, et al. “Flexible modeling of policy-driven upstream notification strategies”. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM. 2014, pp. 1352–1354.

[10] Stefan Craß et al. “Modeling a Flexible Replication Framework for Space-Based Computing”. In: *International Conference on Software Technologies*. Springer. 2013, pp. 256–272.

[11] Alfredo Cuzzocrea. “A query-strategy-focused taxonomy and a customizable benchmarking framework for peer-to-peer information retrieval techniques”. In: *Knowledge-Based Intelligent Information and Engineering Systems*. Springer. 2007, pp. 729–739.

[12] Gianni Di Caro. “Ant Colony Optimization and its application to adaptive routing in telecommunication networks”. PhD thesis. Faculté des Sciences Appliquées, Université Libre de Bruxelles Brussels Belgium, 2004.

[13] Gianni Di Caro and Marco Dorigo. *AntNet: A mobile agents approach to adaptive routing*. Tech. rep. Université Libre de Bruxelles Belgium, 1997.

[14] Gianni A Di Caro. “Analysis of simulation environments for mobile ad hoc networks”. In: *Dalle Molle Institute for Artificial Intelligence, Tech. Rep* (2003).

[15] Alessandro Duminuco, Ernst Biersack, and Taoufik En-Najjary. “Proactive replication in distributed storage systems using machine availability estimation”. In: *Proceedings of the 2007 ACM CoNEXT conference*. ACM. 2007, p. 27.

[16] Cheng Tien Ee et al. “A modular network layer for sensorsets”. In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006, pp. 249–262.

[17] Muddassar Farooq. “A Comprehensive Survey of Nature-Inspired Routing Protocols”. In: *Bee-Inspired Protocol Engineering*. Springer, 2009, pp. 19–52.

[18] Muddassar Farooq. “From The Wisdom of the Hive to Routing in Telecommunication Networks”. In: *Bee-Inspired Protocol Engineering*. Springer, 2009, pp. 53–108.

[19] DS Hickey and LA Noriega. “Insights into information processing by the single cell slime mold physarum polycephalum”. In: *UKACC Control Conference*. 2008, pp. 2–4.

[20] Brian Hollis. “Rapid antagonistic coevolution between strains of the social amoeba Dictyostelium discoideum”. In: *Proceedings of the Royal Society of London B: Biological Sciences* 279.1742 (2012), pp. 3565–3571.

[21] Maarten Houbraken et al. “Fault tolerant network design inspired by Physarum polycephalum”. In: *Natural Computing* 12.2 (2013), pp. 277–289.

[22] Prasad Jogalekar and Murray Woodside. “Evaluating the scalability of distributed systems”. In: *IEEE Transactions on parallel and distributed systems* 11.6 (2000), pp. 589–603.

[23] Daniel Kanev. “Decentralized Unstructured Flat P2P Network with Streaming Content Delivery Method and User Collaboration”. MA thesis. Vienna University of Technology, 2014.

[24] Richard H Kessin. *Dictyostelium: evolution, cell biology, and the development of multicellularity*. Vol. 38. Cambridge University Press, 2001.

[25] Eva Kühn, Stefan Craß, and Thomas Hamböck. “Approaching coordination in distributed embedded applications with the Peer Model DSL”. In: *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*. IEEE. 2014, pp. 64–68.

[26] Eva Kühn et al. “Peer-based programming model for coordination patterns”. In: *International Conference on Coordination Languages and Models*. Springer. 2013, pp. 121–135.

[27] Derek Leonard, Vivek Rai, and Dmitri Loguinov. “On lifetime-based node failure and stochastic resilience of decentralized peer-to-peer networks”. In: *ACM SIGMETRICS performance evaluation review* 33.1 (2005), pp. 26–37.

[28] Bo Leuf. *Peer to Peer: Collaboration and Sharing over the Internet*. Addison-Wesley Professional, 2002.

[29] Eng Keong Lua et al. “A survey and comparison of peer-to-peer overlay network schemes”. In: *Communications Surveys & Tutorials, IEEE* 7.2 (2005), pp. 72–93.

[30] Qin Lv et al. “Search and replication in unstructured peer-to-peer networks”. In: *Proceedings of the 16th international conference on Supercomputing*. ACM. 2002, pp. 84–95.

[31] David Mills. *Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI*. RFC 4330. RFC Editor, 2006, pp. 1–27. URL: `https://tools.ietf.org/html/rfc4330`.

[32] Vinita Mishra and Smita Jangale. “Analysis and comparison of different network simulators”. In: *International Journal of Application or Innovation in Engineering & Management* (2014).

[33] David R Monismith Jr. “Uses of the Slime Mold Lifecycle as a Model for Numerical Optimization”. PhD thesis. Oklahoma State University, 2010.

[34] Sunil Nakrani and Craig Tovey. “On honey bees and dynamic server allocation in internet hosting centers”. In: *Adaptive Behavior* 12.3-4 (2004), pp. 223–240.

[35] Andy Oram. *Peer-to-Peer: Harnessing the power of disruptive technologies*. " O’Reilly Media, Inc.", 2001.

[36] Vidyasagar Potdar, Atif Sharif, and Elizabeth Chang. “Wireless sensor networks: A survey”. In: *Advanced Information Networking and Applications Workshops, 2009. WAINA’09. International Conference on*. IEEE. 2009, pp. 636–641.

[37] Muhammad Saleem, Gianni A Di Caro, and Muddassar Farooq. “Swarm intelligence based routing protocol for wireless sensor networks: Survey and future directions”. In: *Information Sciences* 181.20 (2011), pp. 4597–4624.

[38] Marc Sànchez-Artigas and Enrique Férnandez-Casado. “Evaluation of p2p systems under different churn models: Why we should bother”. In: *European Conference on Parallel Processing*. Springer. 2011, pp. 541–553.

[39] Nurul I Sarkar and Syafnidar A Halim. “A review of simulation of telecommunication networks: simulators, classification, comparison, methodologies, and recommendations”. In: *Journal of Selected Areas in Telecommunications (JSAT), March Edition* (2011).

[40] Vesna Šešum-Čavić, Eva Kuehn, and Daniel Kanev. “Bio-inspired search algorithms for unstructured P2P overlay networks”. In: *Swarm and Evolutionary Computation* 29 (2016), pp. 73–93.

[41] Vesna Sesum-Cavic and eva Kühn. “A Swarm Intelligence Appliance to the Construction of an Intelligent Peer-to-Peer Overlay Network.” In: *CISIS*. 2010, pp. 1028–1035.

[42] Vesna Šešum-Čavić and Eva Kühn. “Self-Organized Load Balancing through Swarm Intelligence”. In: *Next Generation Data Technologies for Collective Computational Intelligence*. Springer, 2011, pp. 195–224.

[43] Daniel Stutzbach and Reza Rejaie. “Understanding churn in peer-to-peer networks”. In: *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*. ACM. 2006, pp. 189–202.

[44] Andrew Tanenbaum and David Wetherall. *Computer Networks Fifth Edition*. Pearson, 2011, pp. 355–494.

[45] Firat Tekiner. “Distributed and intelligent routing algorithm”. PhD thesis. Northumbria University, Newcastle, 2005.

[46] Karl Von Frisch. “Decoding the language of the bee”. In: *Science* 185.4152 (1974), pp. 663–668.

[47] Xiaoming Wang et al. “Robust lifetime measurement in large-scale p2p systems with non-stationary arrivals”. In: *Peer-to-Peer Computing, 2009. P2P’09. IEEE Ninth International Conference on*. IEEE. 2009, pp. 101–110.

[48] Horst F Wedde and Muddassar Farooq. “A performance evaluation framework for nature inspired routing algorithms”. In: *Workshops on Applications of Evolutionary Computation*. Springer. 2005, pp. 136–146.

[49] Horst F Wedde, Muddassar Farooq, and Yue Zhang. “BeeHive: An efficient fault-tolerant routing algorithm inspired by honey bee behavior”. In: *Ant colony optimization and swarm intelligence*. Springer, 2004, pp. 83–94.

[50] Li-Pei Wong, Malcolm Yoke Hean Low, and Chin Soon Chong. “A bee colony optimization algorithm for traveling salesman problem”. In: *Modeling & Simulation, 2008. AICMS 08. Second Asia International Conference on*. IEEE. 2008, pp. 818–823.

[51] Zhongmei Yao et al. “Modeling heterogeneous user churn and local resilience of unstructured P2P networks”. In: *Network Protocols, 2006. ICNP’06. Proceedings of the 2006 14th IEEE International Conference on*. IEEE. 2006, pp. 32–41.

[52] Saloua Zammali and Khedija Arour. “P2pirb: benchmarking framework for p2pir”. In: *Data Management in Grid and Peer-to-Peer Systems*. Springer, 2010, pp. 100–111.

[53] Xiaoge Zhang et al. “An improved physarum polycephalum algorithm for the shortest path problem”. In: *The Scientific World Journal* 2014 (2014).

Web-References

[54] *Apache Commons Net*. http://commons.apache.org/proper/commons-net/. Accessed: 2017-03-23.

[55] *CISCO Internetworking Technology Handbook*. http://docwiki.cisco.com/wiki/Internetworking_Technology_Handbook. Accessed: 2017-03-23.

[56] *Google Compute Engine*. https://cloud.google.com/compute/. Accessed: 2017-03-23.

[57] *Google Compute Engine Machine Types*. https://cloud.google.com/compute/docs/machine-types. Accessed: 2017-03-23.

[58] *J-SIM Website*. https://sites.google.com/site/jsimofficial/. Accessed: 2017-03-23.

[59] *Java Standard Edition 8*. http://docs.oracle.com/javase/8/. Accessed: 2017-03-23.

[60] *jExcel API*. http://jexcelapi.sourceforge.net/. Accessed: 2017-03-23.

[61] *Network Simulator 2*. http://www.isi.edu/nsnam/ns/. Accessed: 2017-03-23.

[62] *Network Simulator 3*. https://www.nsnam.org/. Accessed: 2017-03-23.

[63] *OMNet++ Discrete Event Simulator*. https://omnetpp.org/. Accessed: 2017-03-23.

[64] *PeerfactSim.KOM*. http://peerfact.com/. Accessed: 2017-03-23.

Acronyms

API	Application programming interface
app-data	Application data of a Peer Model entry
BCO	Bee Colony Optimization
cAMP	cyclic Adenosine Monophosphate
co-data	Coordination data of a Peer Model entry
Dd	Dictyostelium discoideum
I/O	Input / Output
ID	Identifier
P2P	Peer-to-Peer
PIC	Peer Input Container
POC	Peer Output Container
Pp	Pseudopod
RT	Routing table
SMNet	SlimeMoldNet
TTL	Time-to-live
TTS	Time-to-start
XLS	Excel Spreadsheet