Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Peer Model Based and Actor Model Based Frameworks for Search Algorithms in Unstructured Peer-to-Peer Networks

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Lukas Fleischhacker
Matrikelnummer 00726030

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, 05.01.2019

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Peer Model Based and Actor Model Based Frameworks for Search Algorithms in Unstructured Peer-to-Peer Networks

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering/Internet Computing

by

Lukas Fleischhacker
Registration Number 00726030

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, 05.01.2019

(Signature of Author) (Signature of Supervisor)

Erklärung zur Verfassung der Arbeit

Lukas Fleischhacker
Pazmanitengasse 28, 1020 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.

(Ort, Datum) (Unterschrift Verfasser)

Danksagung

Als erstes möchte ich mich bei meinen Betreuerinnen Eva Kühn und Vesna Šešum-Cavic für ihre Unterstützung bedanken und für die Möglichkeit, an diesem interessanten Thema zu arbeiten.

Außerdem danke ich Stephan Cejka für seine Hilfe beim Verstehen der Peer Model API.

Ein besonderer Dank gilt auch meinen Freunden für ihr Verständnis und vor allem für die wertvollen Ablenkungen in dieser anstrengenden Zeit. Weiters danke ich meinen ehemaligen Arbeitskollegen Clemens und Oliver, die es mir ermöglicht haben, Beruf und Studium erfolgreich zu vereinigen und mir immer mit wertvollem Rat zur Seite gestanden sind.

Zu guter Letzt möchte ich mich bei meiner gesamten Familie bedanken. Besonders bei meinen Eltern für ihr unermüdliches Verständnis und Vertrauen in mich. Danke, dass ihr immer für mich da seid und ich mich immer auf eure Hilfe und Unterstützung verlassen kann, besonders in schwierigen Zeiten! Ohne euch wäre dieses Studium nicht möglich gewesen!

Abstract

Through its ability to share various resources, P2P networks have been in the focus of research for the last decades. Especially the utilization of search algorithms to retrieve the distributed resources in unstructured P2P networks is of major importance, since no global view of the network exists.

This thesis addresses the need to evaluate and compare search algorithms for unstructured P2P networks with each other by using standard metrics. It provides two frameworks for systematic benchmarking and comparison of search algorithms in unstructured P2P networks. The first framework is implemented based on the Actor Model and the second one based on the Peer Model (a coordination based programming model). Both frameworks support easy exchangeability of search algorithms.

The second goal of this thesis is to create a fully distributed search algorithm for unstructured P2P networks based on the collective feeding of bark beetles. Additionally, an already existing algorithm from a different domain based on the Physarum Polycephalum slime mold is adapted to fully distributed search for unstructured P2P networks.

To achieve these goals the following methodological approach is applied. After an extensive literature research, the frameworks are first designed and afterwards implemented. As the next step, the two new search algorithms are developed and implemented into the frameworks. As the final step, both algorithms are benchmarked, evaluated and compared to four existing search algorithms: Gnutella Flooding, k-Walker, AntNet for P2P and SMP2P.

Overall, Bark Beetle and the adapted Physarum Polycephalum algorithm show very good scalability regarding growing network size and load. In terms of absolute time, both algorithms show very promising results with only k-Walker having slightly better results for networks with high replication. In terms of success rate, Bark Beetle shows an almost equal good success rate as Gnutella Flooding, without having the same major drawbacks. Although the success rate for the Physarum Polycephalum adaption is very low for networks with small replication, it is significantly better for networks with high replication.

Kurzfassung

Aufgrund der Fähigkeit verschiedene Ressourcen zu teilen, sind P2P Netzwerke seit Jahrzehnten im Fokus der Forschung. Vor allem die Umsetzung von Suchalgorithmen, um verteilte Ressourcen in unstrukturierten P2P Netzwerken zu finden, ist von spezieller Wichtigkeit, da keine zentrale Sicht auf das Netzwerk existiert.

Diese Arbeit geht der Notwendigkeit zur Evaluierung und dem Vergleichen von Suchalgorithmen für unstrukturierte P2P Netzwerke, unter der Verwendung von Standardmetriken, nach. Sie liefert zwei Umsetzungen eines Frameworks für systematische Benchmark-Tests von Suchalgorithmen in unstrukturierten P2P Netzwerken und ermöglicht deren Vergleich. Das Framework wird dabei einmal mit dem Actor Model und einmal mit dem auf koordinationsbasiertem Programmiermodel Peer Model umgesetzt. Beide Umsetzungen unterstützen dabei die leichte Austauschbarkeit von Suchalgorithmen.

Das zweite Ziel dieser Arbeit ist das Erstellen eines voll verteilten Suchalgorithmus für unstrukturierte P2P Netzwerke, basierend auf dem kollektiven Futtersuchverhalten von Borkenkäfern. Des Weiteren wird ein bereits bestehender Algorithmus basierend auf dem Physarum Polycephalum Schleimpilz an die Bedürfnisse von voll verteilter Suche in unstrukturierten P2P Netzwerken angepasst.

Um diese Ziele zu erreichen wird folgende methodische Vorgehensweise angewendet. Nach ausführlicher Literaturrecherche werden die Frameworks zunächst entworfen und anschließend umgesetzt. Als nächstes werden die beiden neuen Suchalgorithmen entwickelt und in die Frameworks eingebunden. Abschließend werden beide Algorithmen kompetitiv gemessen, evaluiert und mit vier bekannten Suchalgorithmen verglichen: Gnutella Flooding, k-Walker, AntNet für P2P und SMP2P.

Bark Beetle und der angepasste Physarum Polycephalum Algorithmus zeigen beide insgesamt sehr gute Ergebnisse hinsichtlich der Skalierbarkeit in Bezug auf Netzwerkgröße und -last. Hinsichtlich absoluter Zeit zeigen beide Algorithmen sehr vielversprechende Ergebnisse, wobei lediglich k-Walker etwas bessere Ergebnisse bei Netzwerken mit vielen Replikaten erzielt. In puncto Erfolgsrate zeigt Bark Beetle beinahe so gute Ergebnisse wie Gnutella Flooding, ohne dessen wesentliche Mängel aufzuweisen. Obwohl die Erfolgsrate für die Physarum Polycephalum Anpassung sehr niedrig für Netzwerke mit wenig Replikaten ist, wird diese signifikant besser für Netzwerke mit vielen Replikaten.

Contents

List of Listings

List of Algorithms

*

List of Figures

List of Tables

List of Abbreviations

APP	Application Peer
ASCII	American Standard Code for Information Interchange
cAMP	cyclic Adenosine Monophophate
COP	Coordination Peer
EC	Entry Collection
FID	Flow Identifier
GUID	Globally Unique Identifier
ID	Identifier
IP	Internet Protocol
P2P	Peer-to-Peer
PIC	Peer Input Container
POC	Peer Output Container
RTP	Runtime Peer
SPA	Space Peer
TTL	Time-to-live
URI	Uniform Resource Identifier

CHAPTER 1

Introduction

In this chapter, the problem statement and motivation for the thesis is discussed. Additionally, the aim of the work, the methodological approach to achieve these goals and an overview of the structure of this thesis is provided.

1.1 Problem Statement

A P2P network is a disruptive technology for large scale distributed networking. Due to its ability to share various resources like computer resources, files and network bandwidth without help of any central coordination, it has been in the focus of research for the last decades. One of the main challenges in P2P networks is still the search for resources, since no global view of the network exists and no address mapping is maintained. Thus finding specific resources is neither guaranteed nor bound to a specific upper limit of hops. To address this problem, different non-intelligent and intelligent search algorithms have been proposed. In contrast to non-intelligent search algorithms, intelligent search algorithms learn through the usage of intelligent agents interacting with the environments [15] [13].

Search algorithms need to be evaluated and compared with each other by using standard metrics. Usually algorithms which find more resources in a shorter time by keeping the message load low are rewarded. A popular approach to perform these evaluations is the practice of benchmarking, which ensures the comparability of results in a systematic manner [13].

Therefore, this thesis offers a generic framework that allows plugging in of different search algorithms for unstructured P2P networks. Furthermore, these algorithms will be benchmarked by the provided framework.

Additionally, a new intelligent search algorithm based on the behavior of bark beetles in nature [22] will be created and a comparative analysis to existing fully distributed search algorithms will be performed.

1.2 Aim of the Work

The aim of the thesis is to achieve two goals:

- **Frameworks**: Two frameworks should be implemented for benchmarking and comparison of search algorithms for unstructured P2P networks. The first framework should be implemented based on the Actor Model [16] and the second one based on the Peer Model [12]. Additionally, it should be possible to plug different search algorithms into the frameworks.
- **Bark Beetles and Physarum Polycephalum algorithm**: A new algorithm for distributed search in P2P networks based on the bark beetle's collective feeding [22] should be created. Furthermore, the already existing Physarum Polycephalum algorithm [9] should be adapted for distributed search in P2P. Finally, the created benchmarking frameworks should be used to evaluate the created search algorithms and to compare them to four existing intelligent and non-intelligent search algorithms.

1.3 Methodological Approach

To achieve the defined goals of the thesis, the following approach will be applied:

1. **Literature Research**: The first step is research of literature regarding the technical background and related work of the thesis.
2. **Framework Architecture**: In the next step, the architecture of the frameworks will be designed.
3. **Framework Implementation**: Based on the framework architecture, two frameworks will be implemented using the Java implementation of Akka [1] for the first and the Java implementation of the Peer Model [12] for the second one.
4. **Search Algorithm Development and Adaption**: As a next step, a new search algorithm for P2P called bark beetles for P2P will be developed and the Physarum Polycephalum slime mold algorithm [9] will be adapted for P2P search. Additionally, both algorithms will be implemented into the benchmarking frameworks alongside AntNet [25], SMP2P [27], Gnutella Flooding [21] and k-Random Walker [20].
5. **Benchmarking and Evaluation**: As the final step, the best parameters for each implemented algorithm will be determined following a competitive benchmarking using both frameworks. Based on the results, the performance of the algorithms within and between the two frameworks is evaluated.

1.4 Structure of the Master's Thesis

The structure of this master is as follows:
Chapter 2 provides the technical background of the thesis and discusses the related work. Chapter 3 contains the description of the contributed bark beetles algorithm with detailed implementation in a pseudo code form. Additionally, the Physarum Polycephalum algorithm with its adaption in the context of P2P search is discussed. In Chapter 4 a detailed description of the design and architecture of the benchmarking frameworks is presented. Chapter 5 provides the implementation details of the Akka and Peer Model benchmarking frameworks. Chapter 6 defines the benchmark methodology and provides a performance evaluation and comparative analysis of the presented algorithms within as well as between the two benchmarking frameworks. Finally, Chapter 7 discusses potential improvements to the proposed frameworks and the presented algorithms.

CHAPTER 2

Technical Background and Related Work

This chapter offers the technical background of this thesis and provides an overview of its related work.

2.1 Technical Background

The first part of this section provides a description of the Actor Model and the Peer Model. Afterwards, the basic concepts of P2P networks are described with the main focus on P2P overlays, peer churn and P2P architecture. Finally, two search algorithms for unstructured P2P networks, Gnutella Flooding and the Random walk algorithm, are presented.

2.1.1 Actor Model

The actor model of concurrency was first introduced in 1973. It is a mathematical model of concurrent computation that treats actors as the universal primitives of concurrent digital computation. Its main component is a persistent actor, that encapsulates an internal state and is able to communicate with other actors asynchronously [24] [16].

Actors can communicate with other actors through messages, which is called message-passing and happens asynchronously. For the message-passing no channels or intermediaries are used. Instead, the actor model offers the so-called best effort delivery, which trusts the underlying protocol to deliver the message. Furthermore, it offers at-most-once delivery. Additionally, messages can take arbitrary long to be delivered and no message ordering is guaranteed [16].

When an actor receives a message, it can change the internal state, create more actors, send messages to other actors or respond to the sender zero or more times. Each actor has its own

mailbox, which is a queue of outstanding messages, that the actor processes one at a time, synchronously. Additionally, each actor has an address, which identifies the actor and contains its location and transport information to offer location transparency. An actor system is a collection of actors, their addresses, mailboxes, and configurations. Actor systems and actors have the following characteristics [24] [16]:

- **Communication via direct asynchronous messaging:** To send messages, the sending actor has to know the address of the target actor. The target actor then processes the received message in a separate thread, which allows asynchronous delivery.
- **State machines:** Actors support finite state machines, which allow them to become another kind of message handler by transitioning to another state.
- **Share nothing:** Actors do not share their mutable state with any other actor.
- **Lock-free concurrency:** Since actors never share their mutable state and they only receive one message at a time, actors have no need to be locked and thus never are.
- **Parallelism:** Additionally to concurrency, which describes multiple computations occurring simultaneously, parallelism uses concurrency to achieve a single goal.
- **Actors come in systems:** Parallelism is achieved by using not one but multiple actors.
- **Location Transparency:** Actor addresses are visible to other actors as abstract references and an actor only needs to know that reference to send a message to an actor. The underlying actor system manages the delivery, whether the target actor is on the local actor system or a remote one.
- **Supervision:** Fault-tolerance in the actor model is gained through supervision, where the running state of an actor is monitored and managed by another actor called supervisor, which can perform actions based on the state of the supervised actors. When an error is encountered in a running actor, the default supervising behavior is to restart that actor. Supervision offers a transparent lifecycle management, where addresses do not change during restarts and mailboxes are persisted outside the actor instances, so that no messages are lost during a restart of an actor. Furthermore, supervision strategies can be customized.

2.1.2 Peer Model

The Peer Model was introduced by Kühn et al. in [12] and is a space-based coordination middleware for distributed environments with a data-driven workflow. The strict separation of the coordination from the application logic allows to reuse proven coordination patterns [12].

2.1.2.1 Peer

The main component of the Peer Model is the peer, which represents a structured, reusable, addressable component. Figure 2.1 shows the graphical notation of a peer. Each peer has a name

(URI), an input space called peer-in-container (PIC) and an output space peer-out-container (POC). The peer receives requests via the PIC, takes them out of the PIC to process them and puts the replies into the POC. The actual behavior of peers is realized through sub-peers, wirings and services [12].

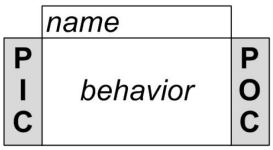


Figure 2.1: Graphical notation of a peer [12]

The Peer Model distinguishes four peer types [12]:

- **Space Peer (SPA):** The SPA is a space container and stores data that is shared between concurrent threads to support communication and synchronization.
- **Coordination Peer (COP):** The COP encapsulates reusable coordination logic like lookup, routing and filtering.
- **Application Peer (APP):** The APP encapsulates application-specific logic in form of service methods.
- **Runtime Peer (RTP):** All hosted peers form a Peer Space, which is bootstrapped via the RTP. The name of this RTP refers to the URI of the local site.

2.1.2.2 Wiring

Wirings are the active part of the system by controlling the movement of entries between the PIC and POC containers. They have a name and consist of three components [12]:

- The guard defines the conditions, when the wiring should activate.
- Optional service calls.
- An action defines how to dispose the resulting entries and consists of zero or more link operations.

The wiring sequentially reads entries via guard links from the PIC and writes them into an entry collection (EC), which is an internal space container for the wiring. All guard links must succeed, if only one fails, the entire guard blocks. Then the wiring performs services on the entries and writes them to destination containers via defined Action Links. As the next step the output links are executed, which distribute the entries of the EC to the PIC and/or POC containers of the own peer or to PIC containers of sub-peers. In contrast to the guard links, not all output links have to succeed. If one fails, it is simply skipped. Finally the effects of

the wiring are automatically committed and the remaining entries are removed from the EC. All wirings whose guards are fulfilled run concurrently and an arbitrary number of instances of the same wiring can run in parallel [12].

2.1.2.3 Flows

The entirety of all wirings that together constitute a business process is called the flow. A flow is identified by a unique flow identifier (FID) and is started by emitting a first entry into the Peer Space with an initial status active. A flow can end through an explicit success of failure result or if it has reached a maximum time-to-live (TTL). Then all entries belonging to this flow will be recognized by wiring and automatically removed. Flows allow architects to model concurrent business processes with the Peer Model [12].

2.1.3 P2P Overview

Stephanos et al. [21] provides the following definition for P2P networks:

"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."

According to this definition, the defining characteristics of a P2P network are fault-tolerance and self-organization as well as the ability to share computer resources by directly exchanging data between nodes without the need of a central server [21].

2.1.3.1 Overlay. Graph Perspective

A P2P network consists of a set of interconnected nodes called peers (e.g. computers) and is built on top of an already existing underlying physical network (e.g. IP network), which is formed independently. Therefore, the P2P network is referred to as an overlay network [21].

The P2P overlay can be represented by an undirected graph `G=(V,E)`, where `V` is the set of peers in the overlay and `E` is the set of connections between the peers. The peers are located in a physical network, which provides mechanisms for a reliable message transport between the peers. Each peer has a unique identification number p_{id}, which identifies it in the overlay network, and a network address n_{id}, which identifies it in the physical network. In the case of an IP network, the n_{id} is the peer's IP address [11].

Let `(p,q)` be an element in `E`, then the following holds [11]:

- `p,q ∈ V`

- p can send a direct message to q over the physical network using q's n_{id} as the destination
- q can send a direct message to p over the physical network using p's n_{id} as the destination
- p and q are called neighbors

The overlay graph G is dynamic, since peers can join or leave the overlay at any given time. An overlay maintenance mechanism ensures that G stays connected. G is a global view of the overlay, whereas each peer has a local view of the overlay called routing table. The routing table of a peer p contains all peers to which it is directly connected, i.e., all edges in G containing p. Each entry in the routing table contains both its p_{id} and n_{id}. If two peers have no direct connection, they can still exchange messages along an existing routing path, which is a sequence of edges from a source peer to a destination peer in G. Each edge that a message traverses is called a hop and thus the path length from source to destination is measured in terms of number of hops. A routing path is recursive, if each successive peer along the path forwards the message to the next peer in the path. A routing path is iterative, if each successive peer along the path replies to the sender with its routing table and the sender is responsible for forwarding the message to the next peer [11].

2.1.3.2 Peer Churn

The dynamic joining and leaving of peers in the P2P overlay networks mentioned in 2.1.3.1 is called peer churn and is a highly significant element of the dynamic nature of P2P networks. The time between the arrival and departure of a peer is defined as a session [10]. Peer churn is one of the key aspects when designing a fault-tolerant and robust P2P network, that can handle unpredictable behavior of peers [5].

2.1.3.3 P2P Architecture

There exist several approaches to classify overlay networks based on different properties. In [21], P2P networks are distinguished based on their centralization and structure.

According to their centralization: In a purely decentralized architecture there is no central coordination unit and thus all peers are servers and clients at the same time and have to perform the same tasks. The nodes in such a network are called servents [21].

A partially centralized architecture is similar to the purely decentralized architecture, but some servents are made supernodes by assigning them that special role based on varying criteria. These supernodes are able to perform additional tasks to support the coordination and collaboration in the P2P network and since they can be dynamically reassigned, they do not constitute a single point of failure [21].

In a hybrid decentralized architecture, one ore more central servers store additional data (e.g. metadata) to help with the coordination and collaboration within the P2P network [21].

According to their structure: Structured P2P networks use global routing tables to provide a mapping between data and peers, which enables deterministic search. Since the mapping has to be updated whenever a peer churn occurs, the efficiency is decreased in highly dynamic environments [15].

Unstructured P2P networks do not have globally maintained mappings between the peers and data. Each peer is responsible for its own data and relies only on its adjacent peers for delivering messages to other peers. Therefore, search has to be performed on incomplete information and thus, the message delivery is not guaranteed. However, unstructured networks perform very good in highly dynamic environments [15].

2.1.4 Search in Fully Distributed P2P Networks

The following section provides an overview of two lookup mechanisms for unstructured P2P networks, Gnutella Flooding and the Random walk algorithm.

2.1.4.1 Gnutella Flooding

Gnutella Flooding consists of a search request Query and a search response QueryHit. When a search query is initiated, a Query message is created, which consists of a unique message ID, the actual search query and a positive integer time-to-live (TTL). Each peer then forwards the Query to all its neighbors and maintains a mapping table between the message ID and the source address of the Query. Additionally, the message ID can be used to prevent potential cycles. At each hop the TTL field is decremented by one and the Query is discarded once TTL equals zero. When matches are found, a QueryHit message is created, which consists of a message ID, that corresponds to the message ID of the Query message, the address of the peer having the matching resource and a list of matching resources. The QueryHit then travels back to the originator taking the same path back as the Query using the mapping table. The Query is still forwarded to all its neighbors, even when a QueryHit message is created [21].

Gnutella Flooding has some drawbacks. First, choosing the appropriate TTL is not trivial, since the success rate drops significantly, if TTL is too low and the networks becomes overloaded, if TTL is too high. Furthermore, duplicate messages result in a serious overhead, which makes it unsuitable for large-scale networks [20].

2.1.4.2 Random walk. k-Walker

The random walk algorithm is an adaption of the flooding approach with the goal to handle the message overhead described in 2.1.4.1. In contrast to flooding, each peer forwards the search query to only one randomly chosen neighbor. Additionally, the search terminates when a result was found instead of forwarding it further. Choosing an appropriate heuristic for selecting the

next random hop is essential. An improvement of random walk is the k-Walker, which starts k parallel search queries [20].

2.2 Related Work

The target area of this thesis is the development and comparison of different intelligent search algorithms for unstructured P2P networks. Therefore, this section provides the description of two existing intelligent search algorithms, Dictyostelium discoideum Slime Mold and AntNet, with their adaption for search in unstructured P2P networks.

2.2.1 Dictyostelium discoideum Slime Mold for P2P

In [27] Šešum-Čavić et al. propose a swarm intelligent algorithm for search in a fully distributed P2P network, which is an adaption of the Dictyostelium discoideum numerical optimization algorithm described in [8]. This section describes the Slime Mold for P2P algorithm (SMP2P) in detail.

In the vegetative stage, amoebae use their pseudopods to explore their neighborhood and move to different positions in search for food. The pseudopods move using an adaption of k-Walker, where the number of neighboring nodes determines the quantity of walkers [27].

Each amoeba waits for its pseudopods to return and applies a probabilistic rule to choose its next hop. While moving through the network, the pseudopods collect local information from each visited node and update their personal best. The amoeba then chooses the node with the best value as its next hop [27].

In order to find better results, amoebae switch into the aggregation stage and cooperate by aggregating around a pacemaker. To ensure that an existing pacemaker can be found, it uses the so-called cyclic Adenosine Monophophate message, which is flooded into the network in a specific radius. When an amoeba first enters the aggregation stage, it first checks for existing pacemakers in its neighborhood. If there is one, it moves toward it, otherwise its personal best is evaluated. If it is greater than one the amoeba terminates its search, otherwise the amoeba is turned into a pacemaker [27].

The aggregated amoebae and their pacemaker form a mound. Amoebae, which improve the personal best of other amoebae in the mound become the head, thus forming a number of slugs [27].

Afterwards, the slug movement is started and the slug moves to the node, where its head found its personal best. At each hop, the slug's amoebae try to improve their personal best. The search can be terminated, if no improvements are found for a certain amount of time [27].

2.2.1.1 AntNet for P2P

In [25] Šešum-Čavić and Kühn presented the AntNet for P2P algorithm, which is an adaption for the context of P2P of the AntNet algorithm designed by Di Caro [14]. AntNet consists of two phases, the forward phase and the backward phase. Starting with the forward phase, forward ants are generated and are launched into the network to gather information about paths and traffic patterns by moving from a source node *s* to a destination node *d*. Once a forward ant reaches its destination node, it is transformed into a backward ant, which in return travels back to s along the same path, but in the opposite direction. The backward ants update the following four data structures at each visited node [25]:

- *Pheromone matrix* T^k: consists of entries τ_{nd}, which is the goodness of the node n for forwarding a package from node k to node d. The following holds:

$$\sum_{n \in N_k} \tau_{nd} = 1, d \in [1, N], \tag{2.1}$$

  where N_k are the neighboring nodes of k and N is the network size.

- *Data-routing table* R^k: has the same structure as T^k. The entries in R^k are computed from the corresponding entries in T^k by exponential transformation, and then normalization to 1. The exponential transformation is used to avoid bad paths.

- *Link queues* L^k: contain the packages waiting to be sent to each neighbor of k and reflect the local network status of the neighborhood of k.

- *Statistical parametric model* M^k: is a list of N-1 triples (μ_d,σ_d^2,W_d). μ_d is the mean and σ_d^2 is the variance of the time required to reach node d from node k. W_d is the best traveling time to d and is computed over a system-wide time window. Similarly to L^k, M^k is a local view of the network traffic.

Each ant maintains a list of already visited nodes and a list of traveling times between the nodes along the traveling path. At each node, the forward ant has to choose its next hop by applying a decision rule. If the forward ant has already visited all neighbors of a node k, the next node is chosen uniformly at random among the neighbors in N_k. To prevent cycles, all nodes involved in a possible cycle have to be removed from the ant's memory. If the node k has some unvisited neighbors, the following formula is applied to choose the following hop [25]:

$$p_{nd} = \frac{\tau_{nd} + \alpha l_n}{1 + \alpha(|N_k| - 1)}, l_n = 1 - \frac{q_n}{\sum\limits_{i \in N_k} q_i} \tag{2.2}$$

where q_i represents the number of bits to be sent to node i and $\alpha \in [0, 1]$.

Once a forward ant reaches its destination node d, it is transformed into a backward ant by only changing its state and preserving all other data structures of the ant. Afterwards, the backward

ant travels back the same route to the source node and updates the statistical model M^k and the pheromone matrix T^k at each visited node k [25].

$$\tau_{fd} = \tau_{fd} + r(1 - \tau_{fd}) \tag{2.3}$$

where f is the node from which the backward ant came to k. To ensure that the pheromone values of all neighbors sum up to 1, the values of the other neighbors are decremented. The reinforcement r can be either a constant, or a function of the variables in the statistical model [25].

To adapt AntNet to the context of P2P Šešum-Čavić and Kühn [26] propose the following:

- P2P nodes are mapped to IP network nodes.
- P2P resources are mapped to destination nodes in the IP network.
- Connections between peers are mapped to IP connections.

Furthermore, the search is adapted to the "not only exact matches" paradigm, by using a similarity function, which describes the quality of the found data [25].

Šešum-Čavić and Kühn [26] show that the performance of AntNet for P2P is significantly better than Gnutella Flooding. In a network of 80 nodes and increasing number of queries the response time of flooding increases exponentially, whereas the response time in AntNet hardly increases.

An adaption for AntNet for P2P to search for similar resources in an unstructured P2P network is presented in [7], [27].

Each node has a pheromone matrix T^k consisting of values τ_{nd}, where n are the neighbors of the node and d is the destination. Since the forward ants carry search queries, the destinations can be represented as query instances. But to support the "not only exact matches" paradigm, the resource found by the forward ant is used as destination [25].

When evaluating the probability in Formula 2.2, the following similarity function compares the destination value to the current search query [25]:

$$p_{nd} = \frac{f(q,d)\tau_{nd} + \alpha l_n}{1 + \alpha(|N_k| - 1)} \tag{2.4}$$

where q is a query, d is a resource in the overlay, which a forward ant has found, f is a similarity function to compare resources and the other variables have the same meaning as in Formula 2.2.

2.2.2 Bee-Intelligence Based P2P Lookup

The algorithm presented by Šešum-Čavić and Kühn [26] uses bee intelligence to search in the P2P context. Bees are represented by software agents, which reside at P2P nodes in the overlay

network. Each node has exactly one hive and one flower. A hive has a number of receiver bees called foragers and outgoing bees called followers. A flower consists of nectar units called tasks, which represent a search task [26].

Starting with the navigation phase, bees are sent into the network to explore it in search for a resource. As soon as the navigation is finished, the bees return to the hive. The following stochastic state transition rule determines the next hop [26]:

$$P_{ij}(t) = \frac{[\rho_{ij}(t)]^{\alpha}[1/d_{ij}]^{\beta}}{\sum_{j \in A_i(t)} [\rho_{ij}(t)]^{\alpha}[1/d_{ij}]^{\beta}} \tag{2.5}$$

where ρ_{ij} is the arc fitness from node *i* to node *j* at time *t* and d_{ij} is the heuristic distance between *i* and *j*, α is a binary variable that turns the arc fitness influence on or off and β is the parameter that controls the significance of a heuristic distance [26].

Afterwards, the recruitment phase begins, in which bees share information, presented by a fitness function [26]:

$$f_i = \frac{1}{H_i}\delta \tag{2.6}$$

where f_i is the fitness value for a particular bee *i*, H_i is the number of hops on the tour and δ is a similarity function, which describes the quality of the found data [26].

The colony's fitness is calculated as follows [26]:

$$f_{colony} = \frac{1}{n}\sum_{i=1}^{n} f_i \tag{2.7}$$

where *n* is the number of outgoing bees. Each bee *i* compares its fitness function f_i with f_{colony} and decides if it becomes a forager or follower [26].

The presented bee algorithm performs especially good in large networks [26].

CHAPTER 3

Algorithms

In this chapter, a definition of the P2P resources that will be used in this thesis is presented as well as two intelligent search algorithms for fully distributed P2P networks. In 3.2 an adaption of the routing algorithm presented by Hickey et. al in [9] based on the Physarum Polycephalum slime mold algorithm is discussed. In 3.3 a search algorithm based on the collective feeding habits of bark beetles is proposed. Both algorithms are contributions to this master thesis.

3.1 P2P Resource Definition

This thesis uses a lookup mechanism for P2P proposed by Šešum-Čavić et al. [27], since it provides the required mechanism to compare similar resources [27].

Let $S = \{D_1, D_2, \dots D_k\}$ be a set of k subsets, where $k \in \mathbb{N}$ and each set defines each element's data type in a resource. Each resource is an ordered n-tuple $r = (r_1, r_2, \dots r_n)$, $n \in \mathbb{N}$, $k \leq n$, $r \in D_{i_1} \times D_{i_2} \times \dots \times D_{i_n}$, where $D_{i_j} \in S$ and $r_i \neq$ **nil**. Search queries are defined equally, but each r_i can be **nil**, which is a zero element. To avoid matches for every resource located in the network, at least one r_i must not be **nil** [27]. The following example illustrates the formalism above and is based on the examples presented in [27].

Example 2.1: A resource representing a movie consists of a movie name, a publishing year and a genre. Each resource consists of the following sets:

- D_1 contains strings up to 255 characters.
- D_2 = $\{x \in \mathbb{N} : 1890 \leq x \leq 2018\}$
- D_3 = {"Drama", "Fantasy", "Science Fiction", "Comedy", "Action"}

The definition of a resource is $r = (r_1, r_2, r_3)$, $r_1 \in D_1$, $r_2 \in D_2$, and $r_3 \in D_3$. The **nil** element in the search query means that any domain value matches. The following shows an example

resource and query:

Resource: ("There Will Be Blood", 2007, "Drama")
Query: ("Blood", **nil**, "Drama")

Additionally, Šešum-Čavić et al. propose in [27] the usage of similarity functions to enable the lookup mechanism to find similar objects. The example above illustrates that the meaning of elements depends on the position of the respective element r_i. Thus, different positions may have different similarity functions [27].

Let $r = (r_1, r_2, \dots r_n)$, $n \in \mathbb{N}$, $r_i \in D_i$, is a resource. δ_i is a similarity function defined as $\delta_i : D_i \times D_i \to \mathbb{R}$ and $(\forall r_i)\ \delta_i(\textbf{nil}, r_i) = 0$. δ_i is in [0,1], where lower values imply higher similarity. The similarity function is defined as follows [27]:

$$f(q, r) = \frac{\sum_{i=1}^{n} \delta_i(q_i, r_i)}{n} \tag{3.1}$$

where $q = (q_1, q_2, \dots q_n)$ represents a query, $r = (r_1, r_2, \dots r_n)$ represents a resource, $n \in \mathbb{N}$, δ_i represents a similarity function for i, and $f \in [0, 1]$ [27].

To determine the quality of matches, a configurable parameter $\epsilon > 0$, $\epsilon \in \mathbb{R}$ is introduced. It categorizes results for a query q and a resource r as follows [27]:

- *no data*: $f(q, r) \geq \epsilon$
- *acceptable data*: $0 < f(q, r) < \epsilon$
- *exact data*: $f(q, r) = 0$

The result set consists of exact and acceptable resource matches [27].

Following **Example 2.1** , similarity functions for movie names (δ_1), publishing years (δ_2) and movie genres (δ_3) are shown below:

$$\delta_1(s_1, s_2) = \begin{cases} 1 - Levenshtein(s_1, s_2), & \text{if } s_2 \neq \textbf{nil} \\ 0, & \text{if } s_2 = \textbf{nil} \end{cases}$$

$$\delta_2(y_1, y_2) = \begin{cases} \frac{|y_1 - y_2|}{2018 - 1890,} & \text{if } y_2 \neq \textbf{nil} \\ 0, & \text{if } y_2 = \textbf{nil} \end{cases}$$

$$\delta_3(j_1, j_2) = \begin{cases} 0, & \text{if } j_1 = j_2 \text{ or } j_2 = \textbf{nil} \\ 0.1, & \text{if } j_1 = \text{“Science Fiction”} \text{ and } j_2 = \text{“Fantasy”} \\ 1, & \text{otherwise} \end{cases}$$

where δ_1 uses the Levenstein string metric [23] and in δ_3 the movie genres “Science Fiction”, and “Fantasy” are similar to each other.

3.2 Physarum Polycephalum Slime Mold for unstructured P2P search

The Physarum Polycephalum slime mold consists of a network of veins, which radially arrange from the centre of the organism as illustrated in Figure 3.1. In search for food, the veins spread out and, if food is found, the respective veins transport the nutrients and dilate to increase their flow. Meanwhile, veins that fail to find food contract and their flow is decreased. Initially, all veins have the same diameter and their flow is considered as equal [9].

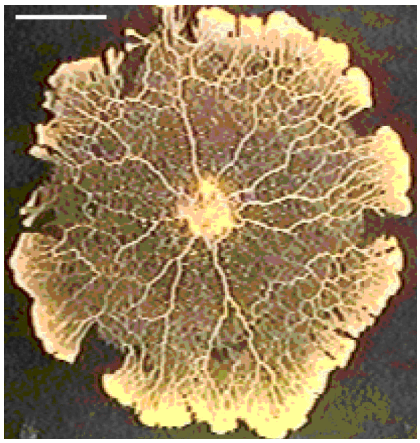


Figure 3.1: Radial arrangement of veins in the Physarum Polycephalum slime mold [9].

[9] Hickey et. al. present a routing algorithm based on the Physarum Polycephalum slime mold. They propose a model similar to the ant colony model [14], since the dilation and contraction of veins is similar to the increase and evaporation of pheromones. The algorithm has a forward phase and a backward phase. During the forward phase nutrients called agents are periodically spawned into the network in search for food. Once an agent finds food, it switches into the backward phase and travels along the same path back, but in the opposite direction. Each node *k* in the network maintains a routing table consisting of entries $Q_{i,j}$, which represent the connecting vein from node *i* to node *j*, i.e. the probability of node *i* of forwarding a data packet from node *k*

to node *j*. The following must hold [9]:

$$\sum_{i \in N_k} Q_{i,j} = 1, j \in [1, N], \tag{3.2}$$

where N_k are the neighboring nodes of *k* and *N* is the network size [9].

Forward agents choose their next hop by applying the following rule [9]:

$$p_{i,j} = \begin{cases} \frac{Q_{i,j}}{\sum\limits_{k \in N_i} Q_{i,j}}, & \text{if } j \in N_i \\ 0, & \text{otherwise} \end{cases} \tag{3.3}$$

where N_i is the neighborhood of *i* and $Q_{i,j}$ is the flow from node *i* to node *j* [9].

Agents in the backward phase, update the routing table of each visited node using the following delta function [9]:

$$\Delta f(|Q_{i,j}|) \leftarrow \xi(Q_{i,j}) \tag{3.4}$$

where $\xi \in [0, 1]$ and ξ should be set to a small value (i.e. < 0.1) [9].

The following presents an adaption of Physarum Polycephalum to the needs of fully distributed P2P networks using the model for a P2P lookup mechanism from 3.1.

The Physarum Polycephalum slime mold for P2P algorithm consists of two phases, the forward phase and the backward phase. Starting with the forward phase, forward agents are launched periodically from the source node s, which initiated the search, into the network to gather information about paths and traffic patterns. Each agent maintains a list of already visited nodes and the search query. Its destination is a resource, that matches the similarity constraints described in 3.1. Each node maintains a flow matrix Q^k consisting of values $Q_{n,d}$, where *n* are the neighbors of the node and *d* is the destination, which is a resource in the overlay, that matches the search query of the forward agent. The following example illustrates this:

Example 3.1: Let the resource ("There Will Be Blood", 2007, "Drama") is located in the network and a search query ("Blood", **nil**, "Drama") is initiated. Suppose the forward agent will eventually find the given resource, an entry for the resource ("There Will Be Blood", 2007, "Drama") is inserted in Q^k.

At each node the forward agent has to choose its next hop by applying a decision rule. If the forward agent has already visited all neighbors of a node *k*, the next node is chosen uniformly at random among the neighbors in N_k. To prevent cycles, all nodes involved in a possible cycle

have to be removed from the agent's memory. If the node *k* has some unvisited neighbors, the next hop is chosen by applying the following decision rule:

$$p_{n,d} = \frac{f(q,d)Q_{n,d}}{\sum\limits_{k \in N_n} f(q,d)Q_{n,d}} \tag{3.5}$$

where *q* is a query, *d* is a resource in the overlay, which a forward agent found and *f* is defined in Equation 3.1.

Once a forward agent reaches its destination, it is transformed into a backward agent by only changing its state and preserving its other data structures. Afterwards, the backward agent travels back the same route to the source node and updates the flow matrix Q^k at each visited node k using Equation 3.4, where a constant value $\in [0, 1]$ is chosen for ξ.

Algorithm 3.1 illustrates the pseudocode of the agent movement.

3.3 Bark Beetle for unstructured P2P search

This section proposes a search algorithm for fully distributed P2P networks based on the collective feeding habits of bark beetles. The first part provides an overview of the general biology of bark beetles. Afterwards, a detailed description of the proposed algorithm is presented.

3.3.1 Bark Beetle in Nature

Bark beetles are a family of beetles with the distinctive characteristic that the parent beetles bore tunnels called galleries under the bark for feeding and egg laying. As shown in Figure 3.2 the life cycle of the bark beetle consists of three phases: reproduction, development, maturation and dispersal [22].

The reproduction begins when mature insects arrive on their host tree, where they construct vertical galleries in which they feed and the female beetle deposits eggs. As soon as the larvae have hatched from their eggs, they start feeding and complete their development in the galleries. Once matured, the beetles disperse to search a suitable host for reproduction. Attacking beetles are often in danger of drowning in their galleries by tree pitch, which is used by the host tree as a defense mechanism. Since beetles are more tolerant of pitch produced by their natural host trees than they are of pitch from other tree species, the effect of pitch influences their choice of a host tree. Bark beetles communicate through two kinds of pheromones, which they excrete while feeding. The attractant pheromones attract additional beetles to the tree to overwhelm its defenses. The anti-attractant pheromones prevent a single tree from being overpopulated [22].

In the following section an algorithm for distributed search for unstructured P2P networks based on the bark beetle's collective feeding, namely the dispersal and communication, is presented.

Algorithm 3.1: Movement of agent

```
1  if moving FORWARD then
2      if current node has resource matching the search query then
3          switch to BACKWARD mode;
4          if in source node then
5              terminate search;
6          else
7              move to previous node;
8          end
9      else
10         if TTL ≤ 0 then
11             terminate search;
12         else
13             if All neighbors are already visited then
14                 choose next hop uniformly at random;
15             else
16                 choose next hop by applying decision rule 3.5;
17             end
18         end
19     end
20 else
21     Update pheromone concentration in current node by applying Equation 3.4;
22     if in source node then
23         terminate search;
24     else
25         move to previous node;
26     end
27 end
```

3.3.2 Initialization

The bark beetle algorithm starts when a peer called source peer initiates a search query. The source peer creates a configurable number of beetles (beetles_per_query) and launches them into the network. The beetles explore the network and return to the source node carrying the search result. Figure 3.3 shows the beetle object.

Each beetle is identified by a unique beetleId that is initialized to a randomly generated globally unique identifier (GUID) and carries a searchQuery of type Resource. Each node in the network has a unique peerId and the currentNodeId identifies the node where the beetle is at a given moment and the sourceNodeId is the identifier of the source peer. Additionally, the TTL variable is the maximum number of hops a beetle can perform and is initialized to a globally configurable beetle_ttl. Additionally, the different pheromone types attractant_pheromone and anti_attrac-

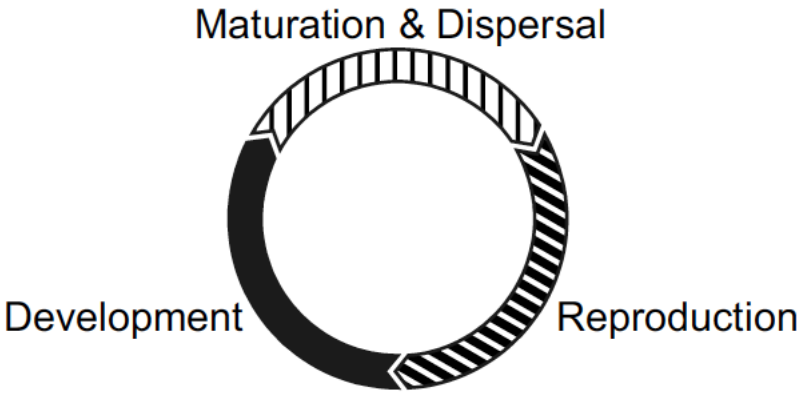


Figure 3.2: Lifecycle of bark beetles [22].

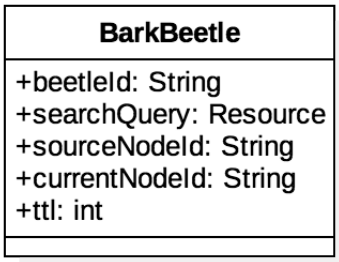


Figure 3.3: UML diagram of the BarkBeetle object

tant_pheromone shown in Table 3.2 are configurable and are initialized to values provided in Table 6.11 in Section 6.2. After the initialization, the beetles start to move.

Table 3.1 summarizes all configurable parameters in the beetle initialization.

Parameter	**Description**
beetles_per_query	The number of beetles per search query
beetle_ttl	The maximum number of hops a beetle can traverse the P2P graph

Table 3.1: Configurable parameters in the beetle initialization

Furthermore, each node maintains a pheromone matrix, which saves the current pheromone concentration for already found resources for all its neighbors. It is initialized to an empty map and is populated during the movement of the beetles, which is described in detail in Section 3.3.3. This pheromone concentration is manipulated by both the attractant_pheromone and anti_attractant_pheromone.

3.3.3 Movement

The beetles move in different directions in search for food by applying the following decision rule. If the pheromone matrix has entries for a resource that matches the search query and the respective pheromone concentration is greater than a configurable sufficient_pheromone_concentration, the node with the highest pheromone concentration is chosen. If no such node exists, the next hop is chosen uniformly at random from its neighbors.

If a beetle finds a resource that matches its search query, it floods the network in a radius R with a notification message called PheromoneMessage, which contains a message identifier messageId, a TTL value ttl, a pheromone value called pheromone and a field foundResource. Figure 3.4 illustrates the PheromoneMessage.

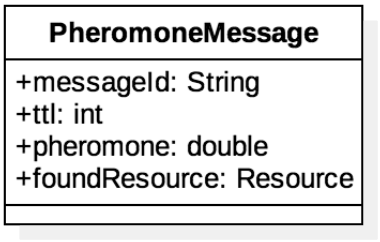


Figure 3.4: UML diagram of the PheromoneMessage object

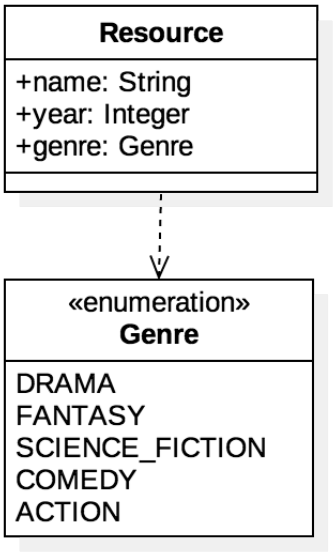


Figure 3.5: UML diagram of the Resource object

The messageId is a randomly generated GUID and is used in the flooding algorithm to prevent processing of already received messages. The variable TTL determines the flooding radius and

is initialized to a configurable pheromone_radius. The variable pheromone is used as the attractant_pheromone and is therefore set to a configurable attractant_pheromone value. Additionally, foundResource saves the found resource illustrated in Figure 3.5. Each node maintains a processedMessages list, which saves the identifiers of already processed notification messages. At each node the PheromoneMessage traverses, its TTL is decreased and its messageId is added to the node's processedMessages list. If the processedMessages list already contains the current messageId, the message is dropped. Otherwise the pheromone matrix of the current node, which saves the overall pheromone concentration of the current node and is manipulated by both the attractant_pheromone and the anti_attractant_pheromone, is updated as follows. If it already contains an entry for foundResource, the entry's current pheromone concentration is incremented by the value of the attractant_pheromone. If no such entry exists, a new entry for the resource in foundResource is created and initialized with the value of the attractant_pheromone. If the TTL value is greater than zero, the PheromoneMessage is forwarded to all the neighbors of the current node. Algorithm 3.2 illustrates the pseudocode of flooding with the PheromoneMessage.

Algorithm 3.2: Flooding with PheromoneMessage

```
1  if the PheromoneMessage was already processed by the current node then
2  |   Drop PheromoneMessage;
3  else
4  |   Decrement TTL;
5  |   Update pheromone matrix of the current node;
6  |   if TTL of the PheromoneMessage > 0 then
7  |   |   Flood neighbors of the current node with the PheromoneMessage;
8  |   else
9  |   |   Drop PheromoneMessage;
10 |   end
11 end
```

Additionally to the flooding, the beetle returns to its source node with the found resource and terminates the search.

At each hop of the beetle, its TTL value is decremented and if it reaches zero or the current node has no more neighbors, the beetle returns to its source node and terminates the search. Furthermore, the beetle checks whether the pheromone matrix of the current node has a pheromone concentration for a resource matching the search query that is greater than a configurable too_sufficient_pheromone_concentration. If such an entry exists, the beetle floods the network using the same mechanism from above (Algorithm 3.2), except that the pheromone value is set to a configurable negative number anti_attractant_pheromone and foundResource is set to the search query.

Table 3.2 summarizes all configurable parameters in the beetle movement.

Parameter	Description
sufficient_pheromone_concentration	The sufficient pheromone concentration to move along the gradient
too_sufficient_pheromone_concentration	The pheromone concentration when the anti attractant pheromone is emitted
pheromone_radius	TTL of the pheromone message
attractant_pheromone	The pheromone concentration that is emitted, if a result is found
anti_attractant_pheromone	The pheromone concentration that is emitted, if the detected pheromone concentration is too high

Table 3.2: Configurable parameters in the beetle movement

Algorithm 3.3 illustrates the pseudocode of the bark beetle movement.

Algorithm 3.3: Movement of bark beetle

```
1  if TTL ≤ 0 then
2      Return to source node and terminate search;
3  else
4      if current node has resource matching the search query then
5          Flood neighbors with attractant_pheromone (Algorithm 3.2);
6          Return to source node with found resource;
7      else
8          if the current node has no neighbors then
9              Return to source node and terminate search;
10         else
11             if A neighbor has a pheromone concentration ≥ too_sufficient_pheromone
                 then
12                 Flood neighborhood with anti_attractant_pheromone (Algorithm 3.2);
13             end
14             Choose the neighbor with highest pheromone concentration;
15             if highest pheromone concentration ≥ sufficient_pheromone_concentration
                 then
16                 Move to neighbor with highest pheromone concentration;
17             else
18                 Choose a random neighbor and move to it;
19             end
20         end
21     end
22 end
```

The sufficient_pheromone_concentration parameter determines whether exploration or exploita-

tion of the search space is performed. At each step, the beetle checks whether a neighbor has a pheromone concentration that is at least equal to the sufficient_pheromone_concentration. If such a pheromone concentration is found, exploitation of the search space is performed. Otherwise the beetle moves randomly and thus is in charge of the exploration of the search space. Additionally, the pheromone_radius determines the number of nodes, which increase their pheromone concentration. Therefore, higher pheromone_radius values foster exploitation. Furthermore, the anti_attractant_pheromone can be used to foster exploration as it decreases the existing pheromone concentration.

CHAPTER 4

Framework Design

This chapter describes the design that will be applied to both frameworks implemented with the Java implementation of Akka and the Java implementation of the Peer Model. The first section gives an overview of the design goals and requirements for the framework. The following sections provide a detailed description of the design of each feature.

4.1 Design goals

The main contribution of this master thesis is the development of a generic framework that allows plugging in of different search algorithms for unstructured P2P networks to compare their respective performance with each other. To achieve this goal, the following high level functional requirements have to be met:

- Generation of different P2P network instances with varying number of peers. Each peer in the resulting network must have at least one direct connection to another peer and must be able to reach any given peer through at least one given routing path. Additionally, it should be possible to distribute the network over different physical devices to decrease the performance load. The number of nodes must be configurable for each test case.
- Distribution of resources among a subset of peers. To allow the "not only exact matches" paradigm the resources should be comparable in such a way that a similarity value between two resources can be calculated. A configurable replication ratio determines the number of peers to which a resource is sent.
- Sending search requests for a specific resource into the network. A configurable parameter determines the satisfying similarity value between two resources and thus, determines when a resource is a valid hit for the issued request. The number of queries must be configurable.

- The actual search is performed by different search algorithms for unstructured P2P networks. It must be possible to plug in different algorithms to perform the search with the respective algorithm.
- A mechanism to determine whether all search requests have been terminated needs to be introduced.
- Finally, the performance of the search has to be evaluated regarding the success rate, the message load per peer and the required time.

4.2 P2P Network Generation

The goal of the P2P graph generation is to create a scale-free and unstructured P2P network. First, a configurable number of peers has to be created. For this, each connected client representing a physical machine is requested to create a certain number of peers, which are resources in the P2P overlay. The following example illustrates this further:

Example 4.1: When a network with 50 nodes should be created and three clients are connected, then client one and client two create each 17 peers and client three creates 16 peers.

To support that different search algorithms can be plugged in, a different peer type for each search algorithm is introduced. Each peer type implements a specific search algorithm, e.g. two distinct peer types exist for Gnutella Flooding and k-Walker. Depending on the selected algorithm, all peers are created of the respective peer type that uses this algorithm to search for resources in the network.

As the next step the neighboring relations between the created peers have to be established. For this the Barabási-Albert (B-A) model is used, which proposes an algorithm to create scale-free networks. In the following, it is used to create a graph, where each node represents a created peer and an edge represents a neighboring relation that has to be established.

The B-A model generates a graph by initializing it with a small number m_0 vertices. Then new vertices are added one by one and each new vertex is connected to $m \leq m_0$ existing vertices. A new vertex is connected to an already existing vertex with the following probability [4]:

$$p_i = \frac{k_i}{\sum_j k_j} \tag{4.1}$$

where k_i is the degree of vertex i, k_j are already existing vertices in the graph and p_i is the probability that a new vertex is connected to vertex i [4].

Following the recommendation in [17], each graph instance is generated with parameters $m = m_0 = 2$.

Finally, the vertices in the created graph are iterated and the neighbors of each vertex are sent to each corresponding peer, which in return establish their neighboring relations.

The network generation process is shown in detail in Figure 4.1.

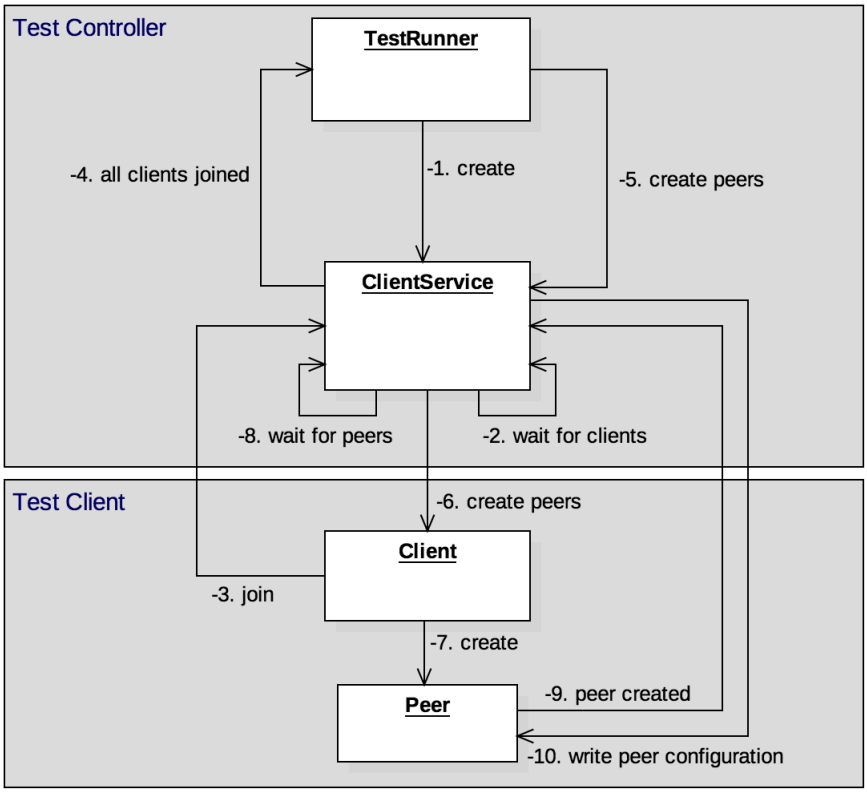


Figure 4.1: P2P Network generation

The TestRunner is the main entity of the Test Controller component and therefore initiates the network generation by creating the ClientService, which is responsible for communicating with the Test Client components. After its creation the ClientService waits for a given number of Clients to join. Therefore, the required Clients have to be started on different machines. After their creation the Clients automatically join the ClientService and as soon as the required number of Clients has joined, the environment is successfully set up and the network can be created.

The TestRunner asks the ClientService to create a given number of peers for the given search algorithm, which in return asks all its joined Clients to create an evenly distributed number of peers on their respective machine. All peers are created of a specific peer type that uses the selected algorithm to search for resources in the network, which provides the functionality to plug in different search algorithms into the framework. Each peer notifies the ClientService about its

creation, which in return waits for all peers to be created. Afterwards, the ClientService is able to directly communicate with all created peers and sends them their respective peer configuration, including their neighboring relations and specific search algorithm parameters.

This approach will be used in both the Akka and Peer Model implementation, with some individual adaptions listed in their respective Sections 4.2.1 and 4.2.2.

4.2.1 Network Generation in Akka

The ClientService, Clients and Peers are all created as actors. After the creation of a Client, it sends a join message to the ClientService. All further communication between the ClientService and the Clients is performed through messages. Each Client creates its peers as depicted in Figure 4.2.

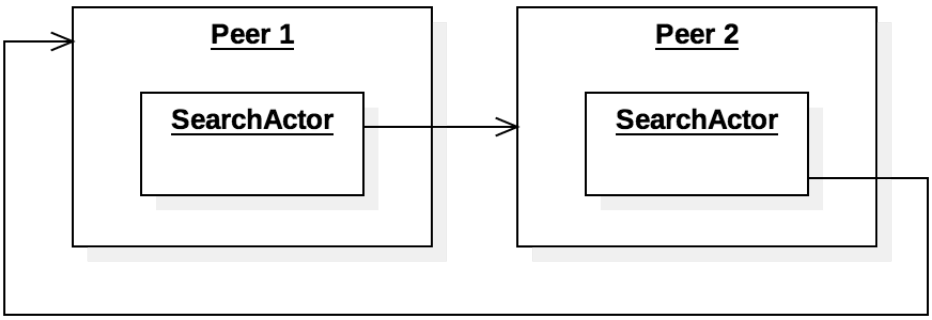


Figure 4.2: Akka Peer

Each peer is an actor and contains a SearchActor that is responsible for the actual search. Depending on the selected search algorithm, different SearchActors are selected. SearchActors cannot communicate directly with each other, instead they send messages to a peer which forwards the message to its SearchActor. Each SearchActor implements the selected search algorithm.

After the peer creation, the peers and ClientService communicate with each other through messages.

4.2.2 Network Generation in the Peer Model

The ClientService is a peer consisting of four sub-peers, namely AddedClients, CreatedPeers, TerminatedSearches and StoppedPeers. Its structure is depicted in Figure 4.3.

Figure 4.3: ClientService Structure

As stated above the ClientService waits after its creation for a given number of Clients to join. Clients are peers consisting of the sub-peer CreatedPeers and a sub-peer for each Node Peer, the Client created. Its general structure is depicted in Figure 4.4.

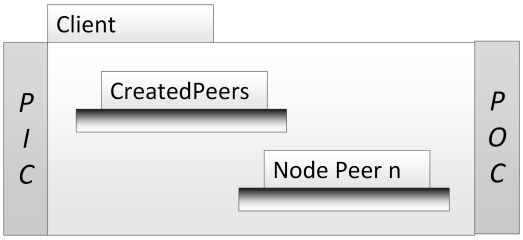


Figure 4.4: Client Structure

After its creation, each Client adds an entry CLIENT_ADDED with its address into the sub-peer AddedClients in ClientService, which in return has a wiring AddedClientWiring depicted in Figure 4.5.

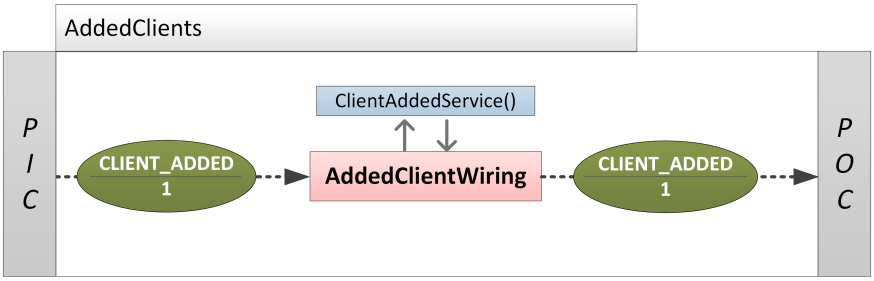


Figure 4.5: AddedClientWiring of the ClientService

Every time an entry CLIENT_ADDED is added to AddedClients, the ClientAddedService adds

the respective Client with its attached address to the joined Clients. As soon as all Clients have joined the ClientService, it notifies the TestRunner.

Afterwards, the ClientService adds entries CREATE_PEER with the selected algorithm attached to the respective Clients, which in return have a wiring CreatingPeersWiring depicted in Figure 4.6.

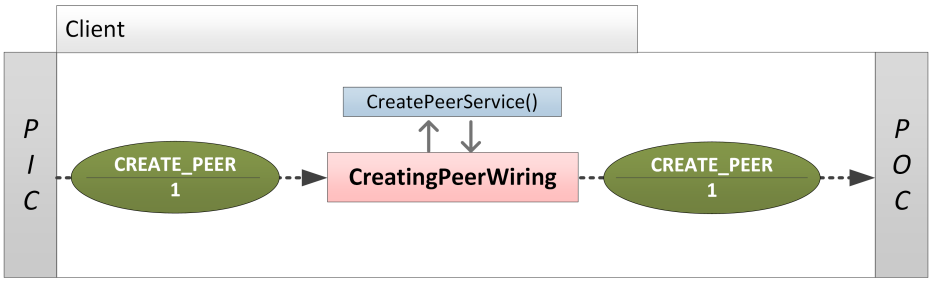


Figure 4.6: CreatingPeersWiring of the Client

Every time an entry CREATE_PEER is added to a Client, the CreatePeerService starts an instance of a new Node Peer with a node type matching the attached algorithm and adds a respective sub-peer to the Client. The created Node Peer instance implements the selected search algorithm, which is described in detail in Section 4.4.

After its creation, each Node Peer adds an entry PEER_CREATED with its address to the sub-peer CreatedPeers in the ClientService, which in return has a wiring PeersCreatedWiring depicted in Figure 4.7.

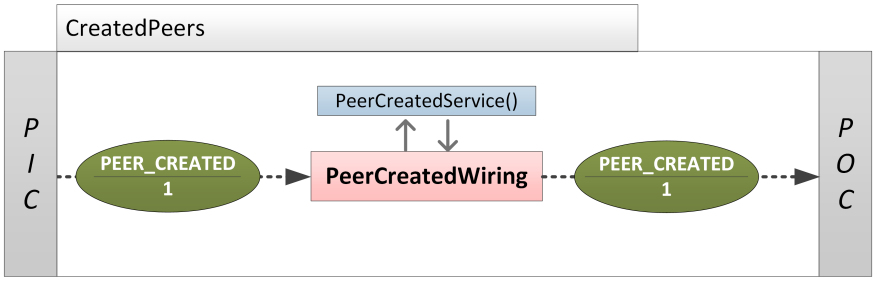


Figure 4.7: PeersCreatedWiring of the ClientService

Every time an entry PEER_CREATED is added to CreatedPeers in the ClientService, the PeerCreatedService adds the respective Node Peer with its address to the ClientService. As soon as all Node Peers are created, the ClientService adds entries ADD_NEIGHBORS to each Node Peer to establish the neighboring relations. Each Node Peer has a wiring AddNeighborsWiring depicted in Figure 4.8.

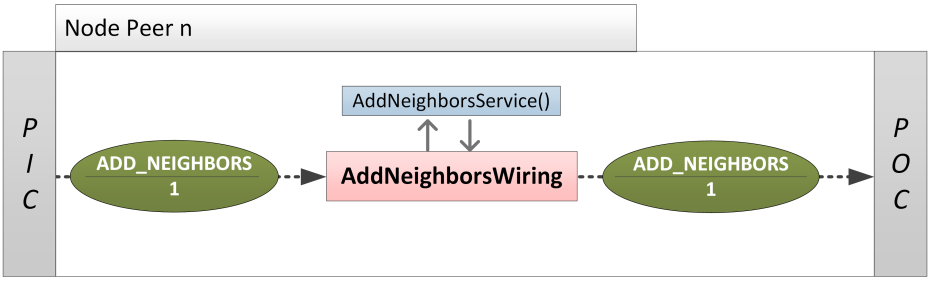


Figure 4.8: AddNeighborsWiring of the Peer Nodes

Every time an entry ADD_NEIGHBORS is added to a Node Peer, the AddNeighborsService adds the neighboring relations to this Node Peer according to the attached neighboring Peer Node addresses. This concludes the P2P network generation process.

4.3 Resource Distribution

Before resources can be distributed among the peers in the created network, the respective resources have to be created. Each resource consists of the following three fields:

- name
- year
- genre

A resource is only valid, if a value is assigned to every field. To make resources comparable, each field uses the similarity functions presented in Section 3.1. The overall similarity is the average of the three single similarities.

Afterwards, the created resources are distributed into the P2P network using the following mechanism. Each resource is sent to certain number of peers, which is determined by a configurable replication ratio parameter. The replication ratio is relative to the network size and thus the number of resources increases for larger networks. The following example clarifies this concept:

Example 4.2: Let a network consisting of 50 nodes, a replication ratio of 16% and two resources are given. Then each resource will be sent to eight peers.

The replication ratio is declared as a value between 0 and 1. Additionally, a replication ratio value of -1 indicates that each resource must only be sent to exactly one peer no matter the size of the network. This paradigm is introduced to simulate a worst case scenario for the search presented in Section 4.4.

Furthermore, each resource is sent to randomly selected peers and one resource can only be sent to a peer exactly once. However, different resources can be sent to the same peer. Following Example 4.2, resource one is sent to eight different peers and resource two is sent to eight different peers, but it is possible that resource one and two are sent the same peer.

The resource distribution process is shown in detail in Figure 4.9.

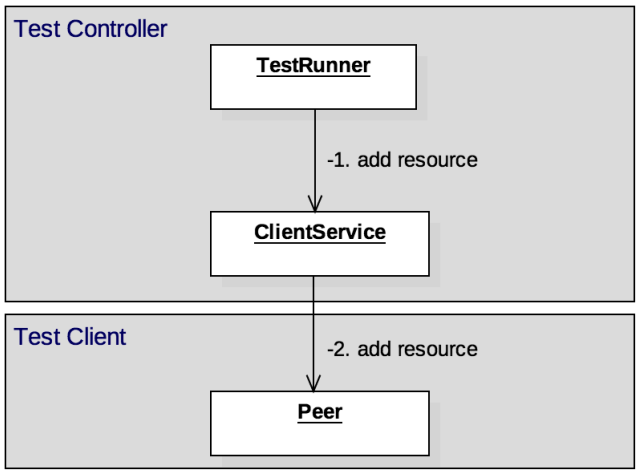


Figure 4.9: Resource Distribution

As stated in Section 4.2, the ClientService is able to communicate directly with all Peers in the created network after the network generation. Therefore, the TestRunner asks the ClientService to push a specific resource into the network. The ClientService in return forwards the resource to a randomly selected peer.

This approach will be used in both the Akka and Peer Model implementation, with some individual adaptions listed in their respective Sections 4.3.1 and 4.3.2.

4.3.1 Resource Distribution in Akka

The actor ClientService sends the resource as a message to the randomly selected peer actor. Since the SearchActor is responsible for the actual search for resources, the message is propagated to the peer's SearchActor, which in return saves the resource.

4.3.2 Resource Distribution in the Peer Model

The ClientService adds an entry ADD_RESOURCE to the respective Node Peer, for each new resource. The Node Peer has a wiring AddResourceWiring depicted in Figure 4.10.

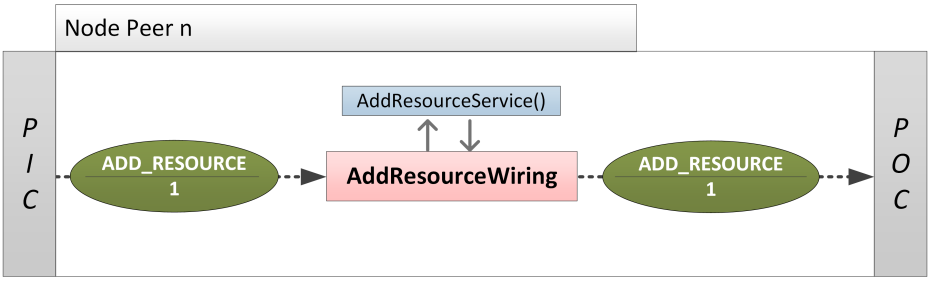


Figure 4.10: AddResourceWiring of the Peer Nodes

Every time an entry ADD_RESOURCE is added to a Node Peer, the AddResourceService adds the attached resource to this Node Peer for later retrieval.

4.4 Search Request

In this step, a configurable number of queries is sent into the P2P network. Similar to the resource distribution in Section 4.3, the number of queries is relative to the network size and is therefore determined through a percentage value. The following example illustrates this concept:

Example 4.3: Let a network consisting of 50 nodes and a query rate of 30% are given, then 15 queries will be issued into the network.

This mechanism ensures that the scalability of the different search algorithms can be examined, since the number of queries grows with increasing network sizes.

Afterwards, permutations of the created query will be generated by creating groups of five instances of the same query, where queries from different groups have different content. This mechanism supports the testing of the "not only exact matches" paradigm. The following example extends **Example 4.3** and illustrates the whole concept of the query generation:

Example 4.4: Let a network consisting of 50 nodes, a query rate of 30% and a query ("Blood", **2007**, "Drama") are given. Then three groups of queries are created. The first group contains five instances of the query ("Blood", **2007**, "Drama"), the second group contains five instances of the query ("Blood", **2008**, "Drama") and the third group contains five instances of the query ("Blood", **2009**, "Drama"). Overall 15 queries are generated, ready to be sent into the network.

After all queries have been created, each query is sent to a randomly selected peer and, like in the query distribution in Section 4.3, all queries are sent to different peers.

The search request process is shown in detail in Figure 4.11.

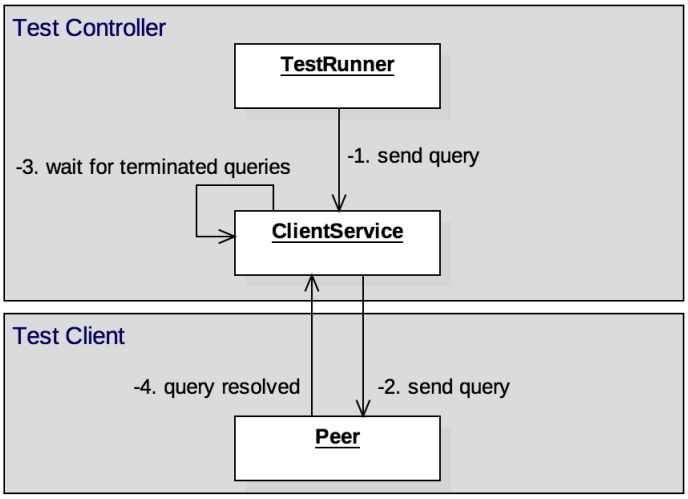


Figure 4.11: Search Request

Similar to the resource distribution in Section 4.3, the TestRunner asks the ClientService to issue specific search requests into the network. In return the ClientService forwards each search request to a randomly selected peer and waits until all queries are terminated.

From this point forward, only the created peers are involved in the search without the intervention or help of the ClientService. Each peer maintains a log file and saves information of the number of query processings and information, if a query can be resolved on this peer with the respective elapsed time or if the query has to be dropped on this peer. Each peer notifies the ClientService when a query terminates and if each query is terminated, the search is completed.

This approach will be used in both the Akka and Peer Model implementation, with some individual adaptions listed in their respective Sections 4.4.1 and 4.4.2.

4.4.1 Search Request in Akka

Similar to the resource distribution in Akka in Section 4.3.1 the actor ClientService sends search requests as a message to randomly selected peers. The peers forward the message to their respective SearchActor, which in return uses the selected search algorithm to search for a resource in network that matches the issued search request by communicating exclusively with other peers. Following the peer communication depicted in Figure 4.2, SearchActors cannot communicate directly with each other. Instead SearchActors send messages to other peers, which forward the message to their respective SearchActor. When a SearchActor finds a matching resource or the query is dropped, it sends a notification message to the ClientService, informing it about the termination of the respective query.

4.4.2 Search Request in the Peer Model

To initiate a search request, the ClientService adds an entry SEARCH_REQUEST with the respective search criteria attached to a selected Node Peer, which in return has a wiring AddSearchRequestWiring depicted in Figure 4.12.

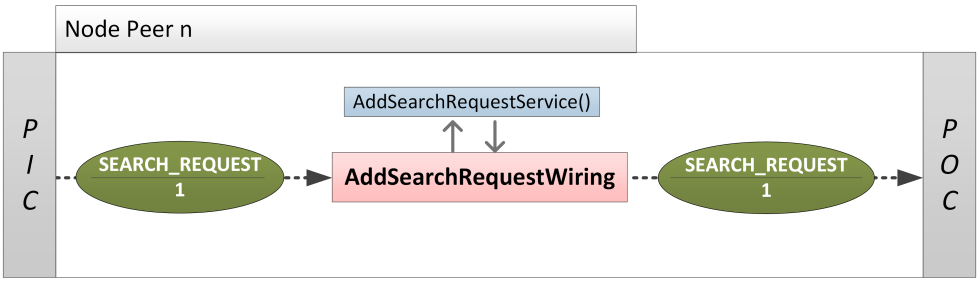


Figure 4.12: AddSearchRequestWiring of the Peer Nodes

Every time an entry SEARCH_REQUEST is added to a Node Peer, the AddSearchRequestService initiates the search according to the attached search criteria in the received entry. From this point forward, only the Node Peers are involved in the search using the selected search algorithm. Figure 4.13 illustrates the communication of the Node Peers during the search.

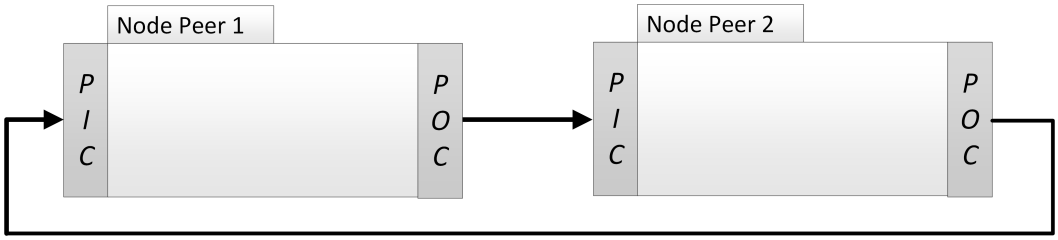


Figure 4.13: Communication of the Peer Nodes

As soon as a search request terminates, the current Node Peer adds an entry SEARCH_TERMINATE to TerminatedSearches in the ClientService, which in return has a wiring TerminatedSearchesWiring depicted in Figure 4.14.

Every time an entry SEARCH_TERMINATE is added to TerminatedSearches, the TerminatedSearchesService checks whether all search requests are terminated. If this is the case, the current test case is finished and the test environment can be reset.

4.5 Reset Test Environment

As soon as all queries are terminated, the log files from all peers have to be retrieved by the TestRunner and are saved for the later performance analysis. Afterwards, the original log files

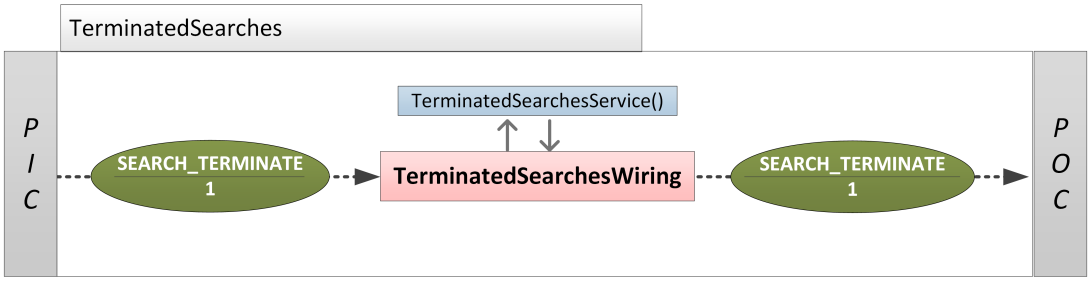


Figure 4.14: TerminatedSearchesWiring of the ClientService

are deleted and the connected clients are issued to destroy their respective peers to reset the test environment to a clean state in order to run each test under the same conditions.

The destruction of the P2P network to reset the test environment to a clean state is shown in detail in Figure 4.15.

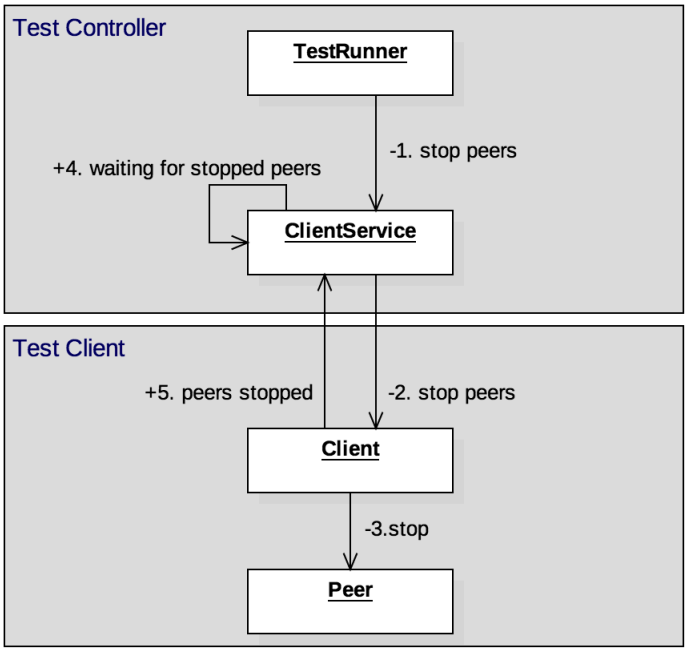


Figure 4.15: Destroy P2P network

The TestRunner issues the ClientService to destroy the generated P2P network by destroying all created peers. The ClientService forwards this request to all connected Clients which destroy their respective peers and notify the ClientService when they are done. As soon as all Clients have confirmed the destruction of their peers, the test environment is successfully reset and the next test case can be executed.

This approach will be used in both the Akka and Peer Model implementation, with some individual adaptions listed in their respective Sections 4.5.1 and 4.5.2.

4.5.1 Rest Test Environment in Akka

Since each peer is an actor, it is sufficient to stop all created peers since all created sub-actors are removed as well in the process. Afterwards each Client sends a notification message to the ClientService, informing it about the successful destruction of its peers.

4.5.2 Reset Test Environment in the Peer Model

To reset the test environment all Node Peers in the network have to be destroyed. For this the ClientService adds an entry STOP_PEERS to all joined Clients, which have a wiring StoppingPeersWiring depicted in Figure 4.16.

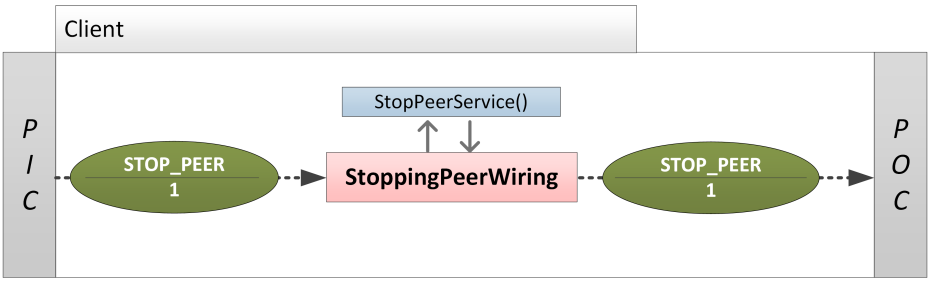


Figure 4.16: StoppingPeersWiring of the Client

Every time an entry STOP_PEERS is added to a Client, the StopPeersService destroys all its Node Peers by removing them from the Client's PeerModel. Afterwards, the Client adds an entry PEERS_STOPPED to StoppedPeers in the ClientService, which has a wiring PeersStoppedWiring depicted in Figure 4.17.

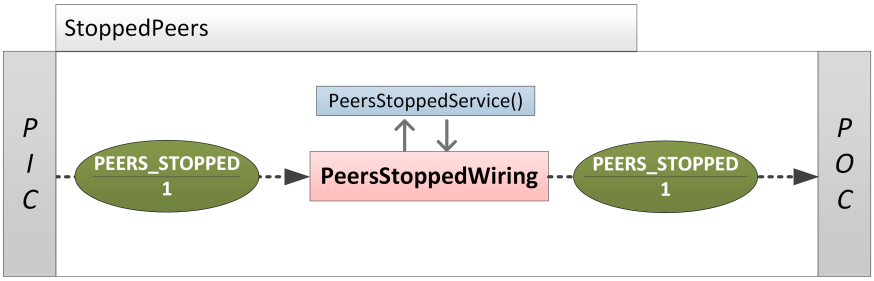


Figure 4.17: PeersStoppedWiring of the ClientService

Every time an entry PEERS_STOPPED is added to StoppedPeers, the PeersStoppedService checks whether all Node Peers are destroyed. If this is the case, the test environment is in a clean state and thus the next test case can be peformed..

CHAPTER 5

Implementation Details

In this chapter the implementation details of the concepts from Chapter 4 are provided. Each section first describes the common implementation followed by the individual details for the Akka and Peer Model frameworks.

5.1 Test Environment Setup

As described in Chapter 4 the TestRunner is responsible for initiating and coordinating each test case. But before any test cases can be executed, the testing environment, that will be used for all subsequent test cases, has to be set up. Therefore, the TestRunner creates a ClientService on its creation, which main purpose is the establishing and maintenance of the testing environment. As described in Chapter 4 the ClientService is the central component, which newly created Clients join. It keeps track of all joined Clients and notifies the TestRunner as soon as all expected Clients have joined. Although the general functionality of the ClientService is the same in the Akka and Peer Model implementation, the implementation details differ and are presented in their respective Sections 5.1.1 and 5.1.2.

5.1.1 Test Environment Setup in Akka

In the Akka implementation, the ClientService is an actor and thus all communication with it is performed through messages. Furthermore, it is started on a static address, which all Clients have to know in order to join it. Each Client is an actor as well and is provided with the address of the ClientService as well as a directory path, where the subsequent test results should be saved. After its creation, each Client sends an AddClientRequest to the ClientService, which in return maintains a list of all joined Clients. Since the address of the sending actor is also appended to the received message, the ClientService can keep track of all Clients, which already sent an AddClientRequest. The TestRunner frequently asks the ClientService whether a certain number of Clients have joined. If this is the case, the testing environment is successfully set up and the TestRunner can proceed with the execution of the test cases.

5.1.2 Test Environment Setup in the Peer Model

In the Peer Model implementation, the ClientService starts a Peer Model instance and adds four sub-peers to it (AddedClients, CreatedPeers, TerminatedSearches and StoppedPeers) illustrated in Figure 4.3. Additionally, a wiring is added to each of these sub-peers. Each Client is provided with the address of the ClientService, a client name and a directory path, where it saves all future test results. Upon its creation, each Client creates a separate Peer Model instance, where the provided client name is used as the instance name. Afterwards, each created Client adds an entry CLIENT_ADDED to the sub-peer AddedClients in the ClientService and appends its address to ensure future communication. In the ClientService the wiring AddedClientWiring (Figure 4.5) is triggered and calls the ClientAddedService, which extracts the address of the joining Client and adds it to a list of joined Clients, maintained by the ClientService. As soon as all expected Clients have joined, the ClientService notifies the TestRunner that the testing environment is ready.

5.2 P2P Network Generation

After the testing environment was successfully established, the TestRunner can run the test cases. For this it provides a method runTest with the following signature:

```
1 runTest(int peers, double replication, int queries, int agentsPerQuery,
      ConfigurationDto configuration, String testName);
```

Listing 5.1: Signature of the method runTest in the TestRunner

For each test case a number of peers, a replication ratio of the resources and a number of queries to be issued is provided. The agentsPerQuery parameter is important to keep track of the resolved queries for search algorithms, which use multiple search agents for one query. The configuration parameter contains search specific information, like the name of the used search algorithm, the similarity threshold parameter and individual search algorithm parameters. Additionally, the testName parameter is used to later identify the used test case. The method runTest executes the steps from Sections 4.2 to 4.5. First the P2P network needs to be created. Therefore, runTest calls the method startPeers in the TestRunner, which has the following signature:

```
1 startPeers(String testname, ConfigurationDto configuration, int peerCount);
```

Listing 5.2: Signature of the method startPeers in the TestRunner

The testName is the name of the current test, the configuration contains the search specific parameters and peerCount is the number of peers, which should be created. The implementation details of Akka and the Peer Model differ and are provided in their respective Sections 5.2.1 and 5.2.2.

5.2.1 P2P Network Generation in Akka

In the Akka implementation, the TestRunner sends a message StartPeersRequest illustrated in Figure 5.1 to the ClientService.

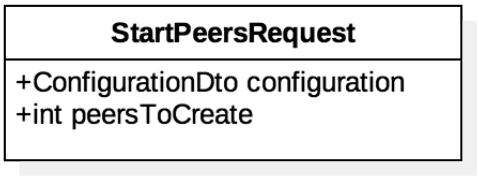


Figure 5.1: UML diagram of the StartPeersRequest message

The ClientService then calculates the number of peers each Client should create. For each new peer, the ClientService sends a message CreatePeerRequest depicted in Figure 5.2 to the respective Client, where id is a globally unique identifier for the new peer and algorithm is the name of the search algorithm, that should be used to retrieve resources.

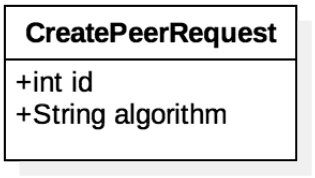


Figure 5.2: UML diagram of the CreatePeerRequest message

Whenever a Client receives a CreatePeerRequest message, it creates a new actor Peer, which is provided with the address of the ClientService, the directory path for the future test results and the peerId and algorithm from the CreatePeerRequest. Upon its creation, the Peer creates another actor called searchActor, which is responsible for the actual search and is provided with the same parameters as the Peer. Depending on the selected search algorithm a different searchActor is created. From this point forward, every message the Peer receives is forwarded to its searchActor, which processes it accordingly. After the searchActor is successfully created it notifies the ClientService about its creation by sending a CreatePeerResponse message. The ClientService maintains a list of created peers and every time it receives a CreatedPeerResponse the sending actor is added to the list. As soon as all peers are created, the ClientService sends a message AddNeighborsMessage depicted in Figure 5.3 to each peer.

It contains an array of neighboring actors for each peer derived from a generated graph using the B-A model and the configuration for search specific parameters. The receiving peer forwards the message to its searchActor, which maintains its neighbors in a list of actors and saves the

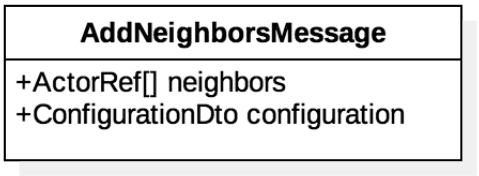


Figure 5.3: UML diagram of the AddNeighborsMessage message

configuration.

Afterwards, the ClientService informs the TestRunner with a list of all created peer actors about the successful creation of the P2P network. From this point forward the TestRunner can sent messages directly to each peer.

5.2.2 P2P Network Generation in the Peer Model

In the Peer Model implementation, the TestRunner calls startPeers in the ClientService, which has the following signature:

```
1 startPeers(ConfigurationDto configuration, int peersToCreate);
```

Listing 5.3: Signature of the method startPeers in the ClientService

It calculates the number of peers for each Client and adds an entry CREATE_PEER for each new Node Peer to the respective Client with the peerId and name of the algorithm attached. In the Client the wiring CreatingPeersWiring (Figure 4.6) is triggered and calls the service CreatePeerService, which extracts the peerId and algorithm name and creates a new Node Peer. Depending on the algorithm a respective SpecialNode is created that extends from SimpleNode, which is depicted in Figure 5.4.

The SpecialNode implements the behavior of the selected search algorithm. Additionally, the SimpleNode adds a new sub-peer with the peerId as its name to the Client, where all future entries for this Node Peer will be added. Additionally, all wirings for the Node Peer will be added to this sub-peer, including AddNeighborsWiring (Figure 4.8), AddResourceWiring (Figure 4.10) and AddSearchRequestWiring (Figure 4.12).

After its creation the Node Peer adds an entry PEER_CREATED with the peerId attached to the sub-peer CreatedPeers in the ClientService, which triggers the wiring PeersCreatedWiring (Figure 4.7). It calls the service PeerCreatedService, which extracts the peerId and adds it to a list of created Node Peers maintained by the ClientService. The ClientService waits for all Node Peers to be created and adds an entry ADD_NEIGHBORS to each Node Peer, with an array of neighboring peerIds, derived from a generated graph using the B-A model, attached. Additionally, the configuration with the search specific parameters is attached. This triggers the

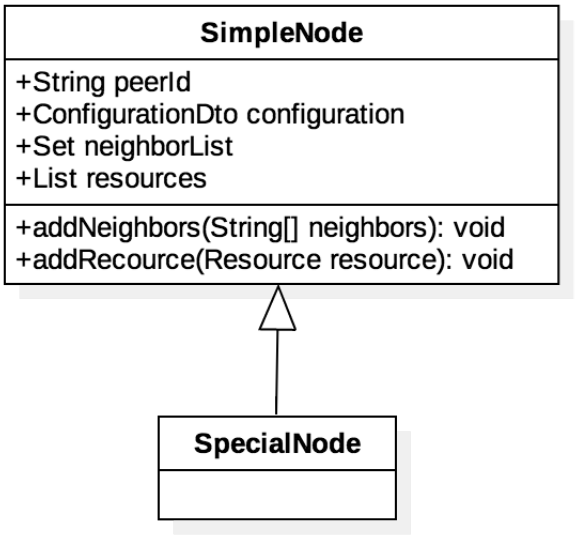


Figure 5.4: UML diagram of SimpleNode

wiring AddNeighborsWiring (Figure 4.8) in the respective Node Peer, which calls the service AddNeighborsService. It extracts the neighbors and configuration and adds both to the SimpleNode, which maintains this information.

Afterwards, the ClientService informs the TestRunner with a list of all created peerIds about the successful creation of the P2P network. From this point forward the TestRunner can add entries directly to each Node Peer.

5.3 Resource Distribution

After startPeers was successfully executed, runTest in the TestRunner calls sendResources with the following signature to distribute a resource among the peers in the network:

```
1 sendResources(String testName, int peerCount, double replicationRatio);
```

Listing 5.4: Signature of the method sendResources in the TestRunner

The testName is the name of the current test, peerCount is the size of the created network and replicationRatio is the percentage that determines to how many peers the resource should be sent. First it creates a Resource object, shown in Figure 5.5.

The Resource consists of a name, a year and a genre. Additionally, it has a method calculateSimilarity that compares itself to another Resource and return their similarity. For this the

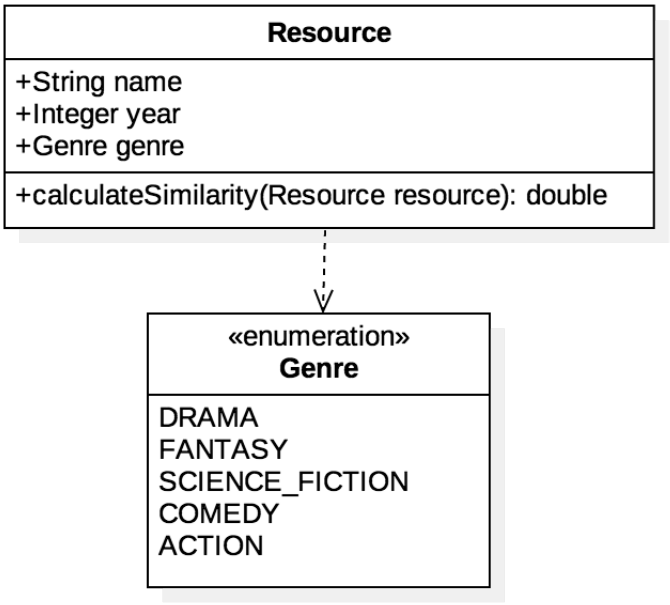


Figure 5.5: UML diagram of Resource

calculateSimilarity method uses the Similarity class depicted in Figure 5.6.

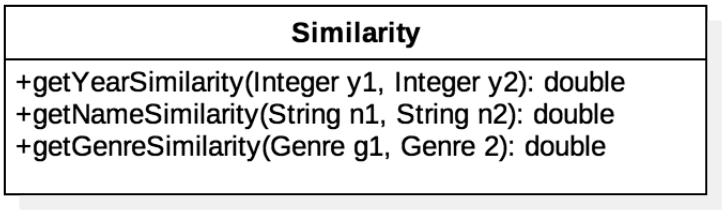


Figure 5.6: UML diagram of Similarity

It has a comparison method for each of the three Resource properties, which use the mechanisms described in Section 3.1.

Afterwards, the created resource is sent to randomly selected peers. The implementation details for Akka and the Peer Model are described their respective Sections 5.3.1 and 5.3.2.

5.3.1 Resource Distribution in Akka

In the Akka implementation sendResources sends a message AddResourceMessage depicted in Figure 5.7 to randomly selected peer actors.

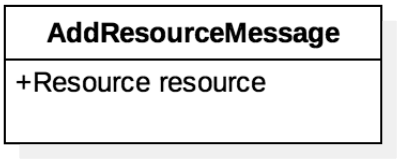


Figure 5.7: UML diagram of the AddResourceMessage message

Each peer forwards the message to its searchActor, which maintains a list of resources.

5.3.2 Resource Distribution in the Peer Model

In the Peer Model implementation sendResources calls the method addResource with the following signature for each randomly selected Node Peer:

```
1 addResource(Sring destination, Resource resource);
```

Listing 5.5: Signature of the method addResource in the ClientService

This method adds a new entry ADD_RESOURCE to the Node Peer destination and attaches the resource. In the respective Node Peer the wiring AddResourceWiring (Figure 4.10) is triggered, which calls the service AddResourceService. This service extracts the resource and adds it to a list of resources, which is maintained by the SimpleNode.

5.4 Search Request

After the resource distribution the runTest method calls sendQueries in the TestRunner with the following signature to issue search requests into the network:

```
1 sendQueries(Sring testName, int peerCount, int queryCount);
```

Listing 5.6: Signature of the method sendQueries in the TestRunner

The testName is the name of the current test, peerCount is the size of the network and queryCount is the number of queries that should be issued. This method creates queries according to the mechanism presented in Section 4.4, whereby each query is a Resource object.

Afterwards, a SearchRequest object is created from each query as depicted in Figure 5.8.

It contains a unique requestId and the actual request as a Resource object. Each SearchRequest object is then sent to randomly selected peers. The implementation details for Akka and the Peer Model are described their respective Sections 5.4.1 and 5.4.2.

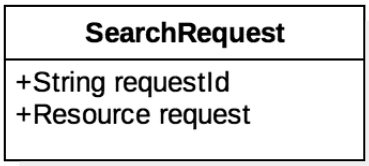


Figure 5.8: UML diagram of the SearchRequest message

5.4.1 Search Request in Akka

In the Akka implementation sendQueries sends the SearchRequest objects as messages to randomly selected peer actors. Each peer forwards the message to its searchActor, which in return applies its search algorithm to search for resources matching the issued query. When the query is either successfully resolved or dropped, the current peer sends a SearchTerminateMessage message depicted in Figure 5.9 to the ClientService.

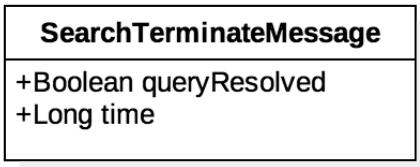


Figure 5.9: UML diagram of the SearchTerminateMessage message

It contains a boolean flag, whether the query was successful and a time value that saves the elapsed time in milliseconds from issuing until termination of a query. The ClientService keeps track of the SearchTerminateMessage messages and compares it to the issued queries to determined whether all queries are terminated.

5.4.2 Search Request in the Peer Model

In the Peer Model implementation sendQueries calls the method sendSearchRequest in the ClientService with the following signature for each randomly selected Node Peer:

```
1 sendSearchRequest(String destination, SearchRequest searchRequest);
```

Listing 5.7: Signature of the method sendSearchRequest in the ClientService

This methods adds a new entry SEARCH_REQUEST to the Node Peer destination and attaches the SearchRequest object. In the respective Node Peer the wiring AddSearchRequestWiring (Figure 4.12) is triggered, which calls the service AddSearchRequestService. This service extracts the SearchRequest object and initiates the search on the current node. From this point

forward solely the Node Peers are involved in executing the selected search algorithm to find a matching resource. When the query is either successfully resolved or dropped, the current Node Peer adds an entry SEARCH_TERMINATE to the sub-peer TerminatedSearches in the ClientService and attaches a boolean flag whether the query was successful and a time value with the elapsed time in milliseconds from issuing until termination of the query. In the ClientService the wiring TerminatedSearchesWiring (Figure 4.14) is triggered, which calls the service TerminatedSearchesService. This service compares the terminated searches with the issued queries to determined whether all queries are terminated.

5.5 Reset Test Environment

After all queries are terminated, the runTest method calls stopPeers in the TestRunner with the following signature to reset the test environment:

```
1 stopPeers(String testname);
```

Listing 5.8: Signature of the method stopPeers in the TestRunner

The implementation details for Akka and the Peer Model are described in their respective Sections 5.5.1 and 5.5.2.

5.5.1 Rest Test Environment in Akka

In the Akka implementation the TestRunner sends a message StopPeersRequest to the ClientService, which forwards the message to all Clients. Each Client stops all its created peers and sends a message StopPeersResponse to the ClientService afterwards, notifying it about the successful destruction of all its peer actors. As soon as all Clients have notified the ClientService, the TestRunner is notified about the successful reset of the test environment and the next test case can be executed.

5.5.2 Reset Test Environment in the Peer Model

In the Peer Model implementation stopPeers calls the method stopPeers in the ClientService with the following signature:

```
1 stopPeers();
```

Listing 5.9: Signature of the method stopPeers in the ClientService

This method adds a new entry STOP_PEER to each Client, where the wiring StoppingPeersWiring (Figure 4.16) is triggered, which calls the service StopPeerService. This service iterates over its Node Peers and removes the respective sub-peer from the Client's Peer Model. Afterwards, an entry PEERS_STOPPED is added to the sub-peer StoppedPeers in the ClientService,

which triggers the wiring PeersStoppedWiring (Figure 4.17). It calls the service PeersStopped-Service, which checks if all Clients have destroyed their respective Node Peers and notifies the TestRunner about it. Afterwards, the test environment is successfully reset and the next test case can be executed.

CHAPTER 6

Evaluation

In this chapter, the Physarum Polycephalum Slime Mold search algorithm adaption and the proposed Bark Beetle for unstructured P2P search algorithm are analyzed and compared to four existing intelligent and non-intelligent search algorithms.

6.1 Simulation Methodology

For the comparison, the search algorithms Gnutella Flooding, k-Walker, AntNet for P2P and SMP2P are benchmarked alongside the Physarum Polycephalum Slime Mold for unstructured P2P search and Bark Beetle for unstructured P2P search algorithms. Additionally, the performance of the algorithms within and between the two proposed frameworks is evaluated.

The topology of the unstructured P2P network is a scale-free network that is generated using the Barabási-Albert (B-A) model [4]. The B-A model initializes the network with m_0 nodes and each new node is connected to $m \leq m_0$ existing nodes. Following the recommendation in [17], each graph instance is generated with parameters $m = m_0 = 2$.

The following dimensions have been identified to perform the benchmarking of the respective algorithms:

- **The P2P graph** follows the definition in Section 2.1.3.1. In the scope of this thesis, only static P2P graph instances without dynamic peer churn are evaluated. Three different network sizes will be used for the benchmarking: 50 nodes, 100 nodes and 200 nodes.
- **The query distribution** is defined as the number of queries that are sent into the P2P network. To represent different load scenarios the number of queries will be distinguished into four groups relative to the network size, covering low to very high load: 10%, 30%, 60%, 90%, i.e. in a network with 50 nodes, 10% means that 5 queries will be used. Since the number of queries is relative to the network size, scalability can be examined.

- **The replication** defines the number of nodes in the P2P network having a specific resource. Similar to the query distribution, the replication ratio of a specific resource is relative to the network size. In the benchmarking, only a single resource is used which is distributed with two different replication strategies. Firstly, to observe the algorithm behavior in the worst case scenario, only one node will have the resource. Secondly, to increase the probability of success, the resource is distributed to 16% of the number of nodes of the network, i.e. 8 replicas for 50 nodes, 16 for 100 nodes and 32 for 200 nodes.

The execution of each test case consists of the following steps:

- **Network generation:** An instance of a P2P network with a given number of nodes is generated using the B-A model and neighboring relationships are established.
- **Configuration:** The configuration settings are sent to each peer, containing the respective parameter settings of the used algorithm.
- **Replication:** An instance of a P2P resource following the definition in Section 3.1 is created and distributed into the P2P network. Depending on the test scenario, the resource is either sent to one node or to 16% of the nodes in the network.
- **Query generation:** Depending on the test scenario, a specific number of queries is created. Additionally, the created queries are divided into groups of five instances of the same query. For example, if 15 queries are created, three groups of five instances of the same query are created, each group containing different content. Afterwards, the created queries are sent to randomly selected nodes. Furthermore, the parameter ϵ is set to a value that the similarity between all queries and available resources results in "acceptable data".
- **Execution:** Each test case is executed ten times and the recorded results are averaged and stored for future analysis.

All benchmarks are carried out in the Google Compute Engine cloud infrastructure [2]. For this, a "n1-standard-16" instance is used, on which the Ubuntu 17.10 operating system is run. The instance includes 16 vCPUs and 60GB RAM. According to [3], 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 following metrics are used for the evaluation:

- **Percentage of successful queries:** A query is successful, if it returns exact or acceptable data, following the definition in Section 3.1. This metric represents the percentage of all sent queries per test case, that were successful.
- **Average messages per node:** This metric represents the average message load for each node in the network.
- **Absolute time:** This metric represents the elapsed time in milliseconds from sending the queries until the queries are resolved.

6.2 Sensitivity Analysis

Before the competitive benchmarks for the different search algorithms can be performed, the best parameter settings for each algorithm have to be evaluated. For this the Sensitivity Analysis is performed on the value range of parameters recommended by previous research works or determined in preliminary benchmarks. Thus, the Sensitivity Analysis is only applied on parameters, if no fixed parameter value is recommended by previous research works. It determines the optimal value for each combination of the different dimensions stated in Section 6.1 based on the metrics in Section 6.1.

To perform the Sensitivity Analysis an automatic parameter tuning mechanism is chosen, specifically the racing mechanism presented in [6]. It focuses only on well performing configurations and discards those that do not perform well enough. Initially, new configurations are run against a small subset of the testing instances. Only the configurations, which do not perform significantly worse than the best yet found configuration, get chosen for the subsequent runs. In each iteration the number of testing instances is increased until only the best configuration is left or a maximum number of test runs is reached. Thus, the more promising a configuration is, the more tests it was run against. In order to effectively compare the candidate configurations the pairwise t-test is used [18].

The Sensitivity Analysis for Gnutella Flooding is performed on its TTL parameter. The TTL value range is shown in Table 6.1 and the analysis results are shown in Table 6.2.

Parameter	**Range**	**Source**
TTL	7, 8, 9	preliminary benchmarks

Table 6.1: Gnutella Flooding parameter values for the Sensitivity Analysis

nodes	**queries**	**TTL**
50	5	9
50	15	
50	30	
50	45	
100	10	
100	30	
100	60	
100	90	
200	20	
200	60	
200	120	
200	180	

Table 6.2: Gnutella Flooding Sensitivity Analysis results

The Sensitivity Analysis for k-Walker is performed on the number of walkers and the TTL parameter. The parameter range for the walkers is based on the recommendation in [20] and the

TTL parameter range is based on preliminary benchmarks. The parameter value ranges for the Sensitivity Analysis are shown in Table 6.3 and the analysis results are shown in Table 6.4.

Parameter	**Range**	**Source**
walkers	16, 32	[20]
TTL	7, 8, 9	preliminary benchmarks

Table 6.3: k-Walker parameter settings for the Sensitivity Analysis

nodes	**queries**	**walkers**	**TTL**
50	5	32	9
50	15		
50	30		
50	45		
100	10		
100	30		
100	60		
100	90		
200	20		
200	60		
200	120		
200	180		

Table 6.4: k-Walker Sensitivity Analysis results

The Sensitivity Analysis for AntNet for P2P is only performed on the TTL parameter and the remaining parameter values are chosen based on the recommendations in [25]. The parameter value range for TTL is based on preliminary benchmarks. The parameter value ranges for the Sensitivity Analysis is shown in Table 6.5 and the analysis results are shown in Table 6.6.

Parameter	**Range**	**Source**
TTL	4, 7, 9	preliminary benchmarks
α	0.2	[25]
C_2	0.25	

Table 6.5: AntNet for P2P parameter values for the Sensitivity Analysis

All parameters for the SMP2P algorithm are chosen based on the recommendations in [27]. Since they showed good results in the preliminary benchmarks, no Sensitivity Analysis is necessary. The parameter values used for the competitive benchmarks are shown in Table 6.7.

The Sensitivity Analysis for Physarum Polycephalum Slime Mold for unstructured P2P search is performed on the TTL and ξ parameters. The parameter value range for both parameters is based on preliminary benchmarks. The parameter value ranges for the Sensitivity Analysis are shown in Table 6.8 and the analysis results are shown in Table 6.9.

nodes	queries	TTL
50	5	9
50	15	
50	30	
50	45	
100	10	
100	30	
100	60	
100	90	
200	20	
200	60	
200	120	
200	180	

Table 6.6: AntNet for P2P Sensitivity Analysis results

nodes	queries	minimum slug update count	min aggregate count	pacemaker notification ttl	pseudopod max ttl	aggregate count threshold	roulette wheel selection prob.
50	5	60	4	7	4	16	0.2
50	15	60	4	4	4	16	0.2
50	30	15	16	7	4	4	0.2
50	45	30	16	7	4	16	0.2
100	10	60	4	2	4	4	0.8
100	30	15	4	4	4	4	0.2
100	60	15	16	7	4	16	0.2
100	90	30	16	4	4	4	0.6
200	20	60	4	7	7	4	0.8
200	60	30	16	7	7	4	0.6
200	120	60	16	2	4	4	0.2
200	180	30	16	7	4	4	0.2

Table 6.7: SMP2P parameter values used for the competitive benchmarks

Parameter	Range	Source
TTL	7, 8, 9	preliminary benchmarks
ξ	0.025, 0.05, 0.1	

Table 6.8: Physarum Polycephalum Slime Mold parameter values for the Sensitivity Analysis

nodes	queries	TTL	ξ
50	5	9	0.1
50	15		
50	30		
50	45		
100	10		
100	30		
100	60		
100	90		
200	20		
200	60		
200	120		
200	180		

Table 6.9: Physarum Polycephalum Slime Mold Sensitivity Analysis results

The Sensitivity Analysis for Bark Beetle for unstructured P2P search is performed on the TTL, radius, sufficient pheromone rate, too sufficient pheromone rate, attractant pheromone and anti attractant pheromone parameters. The parameter value ranges for those parameters are based on preliminary benchmarks. The parameter value ranges for the Sensitivity Analysis are shown in Table 6.10 and the analysis results are shown in Table 6.11.

Parameter	Range	Source
TTL	7, 8, 9	preliminary benchmarks
radius	1, 2, 3	
sufficient pheromone rate	1, 5, 10	
too sufficient pheromone rate	100, 200, 500	
attractant pheromone	0.25, 0.5, 1	
anti attractant pheromone	-0.25, -0.5, -1	

Table 6.10: Bark Beetle parameter values for the Sensitivity Analysis

nodes	queries	TTL	radius	sufficient pheromone rate	too sufficient pheromone rate	attractant pheromone	anti attractant pheromone
50	5	9	2	1	100	1	-1
50	15		1	1	100		
50	30		1	1	100		
50	45		1	1	100		
100	10		2	1	100		
100	30		1	1	100		
100	60		1	5	200		
100	90		1	5	200		
200	20		1	1	100		
200	60		1	1	200		
200	120		1	5	500		
200	180		1	5	500		

Table 6.11: Bark Beetle Sensitivity Analysis results

6.3 Raw Result Data

The results of each algorithm obtained with the Peer Model framework are shown in the Tables 6.12, 6.13, 6.14, 6.15, 6.16 and 6.17. The results of each algorithm obtained with the Akka framework are shown in the Tables 6.18, 6.19, 6.20, 6.21, 6.22 and 6.23. The columns in all the tables are explained as follows:

- **#:** The row number.
- **# nodes:** The number of nodes.
- **# queries:** The number of issued queries.
- **replication:** The replication ratio of the resource in the P2P network.
- **success rate:** The success rate percentage metric as defined in 6.1.
- **avg message per node:** The average messages per node metric as defined in 6.1.
- **absolute time:** The absolute time metric in milliseconds as defined in 6.1.

#	# nodes	# queries	replication	success rate	avg message per node	absolute time
1	50	5	0.02	100	14	108
2	50	5	0.16	100	17	44
3	50	15	0.02	99	43	179
4	50	15	0.16	100	51	81
5	50	30	0.02	98	85	318
6	50	30	0.16	100	103	185
7	50	45	0.02	98	127	524
8	50	45	0.16	100	152	224
9	100	10	0.01	100	28	263
10	100	10	0.16	100	19	64
11	100	30	0.01	99	81	681
12	100	30	0.16	100	102	297
13	100	60	0.01	100	160	1237
14	100	60	0.16	100	184	592
15	100	90	0.01	99	240	1859
16	100	90	0.16	100	307	1047
17	200	20	0.005	100	53	922
18	200	20	0.16	100	30	226
19	200	60	0.005	95	151	2520
20	200	60	0.16	100	185	1086
21	200	120	0.005	95	295	5021
22	200	120	0.16	100	382	2172
23	200	180	0.005	95	438	7217
24	200	180	0.16	100	579	3753

Table 6.12: Gnutella Peer Model results

#	# nodes	# queries	replication	success rate	avg message per node	absolute time
1	50	5	0.02	96	27	148
2	50	5	0.16	100	20	20
3	50	15	0.02	89	84	411
4	50	15	0.16	100	57	70
5	50	30	0.02	89	163	605
6	50	30	0.16	100	101	100
7	50	45	0.02	89	247	888
8	50	45	0.16	100	146	141
9	100	10	0.01	80	27	193
10	100	10	0.16	100	13	22
11	100	30	0.01	83	80	643
12	100	30	0.16	100	45	88
13	100	60	0.01	84	164	1037
14	100	60	0.16	100	99	176
15	100	90	0.01	83	247	1554
16	100	90	0.16	100	159	273
17	200	20	0.005	41	28	526
18	200	20	0.16	100	15	69
19	200	60	0.005	37	85	1732
20	200	60	0.16	100	51	220
21	200	120	0.005	39	171	2948
22	200	120	0.16	100	100	360
23	200	180	0.005	39	257	4192
24	200	180	0.16	100	160	582

Table 6.13: k-Walker Peer Model results

#	# nodes	# queries	replication	success rate	avg message per node	absolute time
1	50	5	0.02	10	8	9
2	50	5	0.16	100	8	55
3	50	15	0.02	17	28	62
4	50	15	0.16	97	27	178
5	50	30	0.02	19	54	250
6	50	30	0.16	100	43	217
7	50	45	0.02	30	84	504
8	50	45	0.16	98	68	446
9	100	10	0.01	54	9	167
10	100	10	0.16	98	6	56
11	100	30	0.01	19	27	417
12	100	30	0.16	98	20	252
13	100	60	0.01	26	57	1045
14	100	60	0.16	96	48	676
15	100	90	0.01	13	84	748
16	100	90	0.16	97	68	885
17	200	20	0.005	5	9	221
18	200	20	0.16	99	6	147
19	200	60	0.005	4	28	604
20	200	60	0.16	92	23	728
21	200	120	0.005	4	59	1197
22	200	120	0.16	94	46	1479
23	200	180	0.005	15	86	2688
24	200	180	0.16	94	77	2488

Table 6.14: AntNet Peer Model results

#	# nodes	# queries	replication	success rate	avg message per node	absolute time
1	50	5	0.02	44	10	177
2	50	5	0.16	100	17	182
3	50	15	0.02	52	30	426
4	50	15	0.16	92	44	462
5	50	30	0.02	63	76	957
6	50	30	0.16	100	88	920
7	50	45	0.02	66	118	1454
8	50	45	0.16	100	131	1364
9	100	10	0.01	63	18	528
10	100	10	0.16	100	22	413
11	100	30	0.01	65	52	1133
12	100	30	0.16	99	78	1142
13	100	60	0.01	67	105	2243
14	100	60	0.16	95	152	2239
15	100	90	0.01	62	166	3321
16	100	90	0.16	98	218	3374
17	200	20	0.005	33	24	1153
18	200	20	0.16	75	27	1015
19	200	60	0.005	40	62	3365
20	200	60	0.16	79	76	2359
21	200	120	0.005	27	73	5098
22	200	120	0.16	81	92	4497
23	200	180	0.005	28	109	7253
24	200	180	0.16	85	159	6320

Table 6.15: SMP2P Peer Model results

#	# nodes	# queries	replication	success rate	avg message per node	absolute time
1	50	5	0.02	10	9	17
2	50	5	0.16	88	9	72
3	50	15	0.02	6	28	27
4	50	15	0.16	97	23	143
5	50	30	0.02	17	54	75
6	50	30	0.16	93	45	219
7	50	45	0.02	26	84	154
8	50	45	0.16	96	60	313
9	100	10	0.01	26	9	86
10	100	10	0.16	84	6	61
11	100	30	0.01	24	29	192
12	100	30	0.16	82	19	163
13	100	60	0.01	20	59	353
14	100	60	0.16	89	42	420
15	100	90	0.01	33	91	693
16	100	90	0.16	91	73	604
17	200	20	0.005	7	9	80
18	200	20	0.16	84	6	116
19	200	60	0.005	12	29	238
20	200	60	0.16	83	22	449
21	200	120	0.005	7	59	379
22	200	120	0.16	97	46	880
23	200	180	0.005	13	91	547
24	200	180	0.16	90	70	1158

Table 6.16: Physarum Polycephalum Slime Mold Peer Model results

#	# nodes	# queries	replication	success rate	avg message per node	absolute time
1	50	5	0.02	82	8	89
2	50	5	0.16	100	17	45
3	50	15	0.02	100	22	169
4	50	15	0.16	100	15	69
5	50	30	0.02	100	42	289
6	50	30	0.16	100	28	120
7	50	45	0.02	100	63	408
8	50	45	0.16	100	40	164
9	100	10	0.01	100	10	157
10	100	10	0.16	100	7	63
11	100	30	0.01	97	22	437
12	100	30	0.16	100	14	110
13	100	60	0.01	99	45	1229
14	100	60	0.16	100	30	239
15	100	90	0.01	97	67	1399
16	100	90	0.16	100	45	350
17	200	20	0.005	67	7	370
18	200	20	0.16	100	5	115
19	200	60	0.005	78	23	1098
20	200	60	0.16	100	14	257
21	200	120	0.005	69	47	2755
22	200	120	0.16	100	30	521
23	200	180	0.005	73	70	3814
24	200	180	0.16	100	44	788

Table 6.17: Bark Beetle Peer Model results

#	# nodes	# queries	replication	success rate	avg message per node	absolute time
1	50	5	0.02	100	14	13
2	50	5	0.16	100	16	4
3	50	15	0.02	100	44	6
4	50	15	0.16	100	51	3
5	50	30	0.02	100	88	11
6	50	30	0.16	100	102	5
7	50	45	0.02	100	131	53
8	50	45	0.16	100	153	20
9	100	10	0.01	100	29	11
10	100	10	0.16	100	35	12
11	100	30	0.01	100	88	81
12	100	30	0.16	100	107	29
13	100	60	0.01	100	176	54
14	100	60	0.16	100	215	137
15	100	90	0.01	100	264	242
16	100	90	0.16	100	322	77
17	200	20	0.005	100	58	42
18	200	20	0.16	100	72	5
19	200	60	0.005	100	176	377
20	200	60	0.16	100	220	85
21	200	120	0.005	100	348	441
22	200	120	0.16	100	439	182
23	200	180	0.005	99	518	466
24	200	180	0.16	100	658	177

Table 6.18: Gnutella Akka results

#	# nodes	# queries	replication	success rate	avg message per node	absolute time
1	50	5	0.02	86	28	13
2	50	5	0.16	100	21	1
3	50	15	0.02	90	84	23
4	50	15	0.16	100	58	31
5	50	30	0.02	88	162	14
6	50	30	0.16	100	100	16
7	50	45	0.02	86	248	134
8	50	45	0.16	100	146	1
9	100	10	0.01	83	27	2
10	100	10	0.16	100	13	1
11	100	30	0.01	82	80	150
12	100	30	0.16	100	45	46
13	100	60	0.01	85	164	154
14	100	60	0.16	100	100	2
15	100	90	0.01	83	247	80
16	100	90	0.16	100	158	39
17	200	20	0.005	39	28	6
18	200	20	0.16	100	14	94
19	200	60	0.005	35	85	83
20	200	60	0.16	100	51	3
21	200	120	0.005	40	171	883
22	200	120	0.16	100	101	90
23	200	180	0.005	38	257	319
24	200	180	0.16	100	159	12

Table 6.19: k-Walker Akka results

#	# nodes	# queries	replication	success rate	avg message per node	absolute time
1	50	5	0.02	4	9	1
2	50	5	0.16	96	9	3
3	50	15	0.02	28	29	4
4	50	15	0.16	98	23	4
5	50	30	0.02	15	57	11
6	50	30	0.16	95	46	8
7	50	45	0.02	24	89	14
8	50	45	0.16	97	63	12
9	100	10	0.01	26	9	7
10	100	10	0.16	93	5	3
11	100	30	0.01	20	28	15
12	100	30	0.16	93	21	6
13	100	60	0.01	22	61	21
14	100	60	0.16	85	48	17
15	100	90	0.01	35	89	182
16	100	90	0.16	87	74	42
17	200	20	0.005	6	9	1
18	200	20	0.16	88	6	9
19	200	60	0.005	3	29	6
20	200	60	0.16	73	23	28
21	200	120	0.005	0	59	14
22	200	120	0.16	90	46	94
23	200	180	0.005	1	89	281
24	200	180	0.16	85	74	116

Table 6.20: AntNet Akka results

#	# nodes	# queries	replication	success rate	avg message per node	absolute time
1	50	5	0.02	48	15	68
2	50	5	0.16	90	19	100
3	50	15	0.02	70	41	136
4	50	15	0.16	96	54	112
5	50	30	0.02	73	79	247
6	50	30	0.16	97	96	180
7	50	45	0.02	77	122	371
8	50	45	0.16	90	138	318
9	100	10	0.01	76	23	147
10	100	10	0.16	99	21	129
11	100	30	0.01	74	53	286
12	100	30	0.16	87	57	300
13	100	60	0.01	71	110	626
14	100	60	0.16	93	123	583
15	100	90	0.01	57	145	1156
16	100	90	0.16	91	172	824
17	200	20	0.005	43	24	215
18	200	20	0.16	89	37	244
19	200	60	0.005	47	67	464
20	200	60	0.16	84	100	480
21	200	120	0.005	37	90	684
22	200	120	0.16	85	150	757
23	200	180	0.005	22	112	839
24	200	180	0.16	68	178	765

Table 6.21: SMP2P Akka results

#	# nodes	# queries	replication	success rate	avg message per node	absolute time
1	50	5	0.02	12	10	5
2	50	5	0.16	98	7	1
3	50	15	0.02	30	29	28
4	50	15	0.16	95	26	2
5	50	30	0.02	10	58	6
6	50	30	0.16	94	44	4
7	50	45	0.02	2	87	3
8	50	45	0.16	97	56	5
9	100	10	0.01	14	9	2
10	100	10	0.16	90	6	5
11	100	30	0.01	11	28	9
12	100	30	0.16	91	19	10
13	100	60	0.01	26	60	10
14	100	60	0.16	88	44	26
15	100	90	0.01	23	92	200
16	100	90	0.16	94	66	69
17	200	20	0.005	7	9	17
18	200	20	0.16	80	6	7
19	200	60	0.005	7	29	6
20	200	60	0.16	89	23	23
21	200	120	0.005	4	60	3
22	200	120	0.16	88	44	50
23	200	180	0.005	8	89	27
24	200	180	0.16	93	68	71

Table 6.22: Physarum Polycephalum Slime Mold Akka results

#	# nodes	# queries	replication	success rate	avg message per node	absolute time
1	50	5	0.02	92	8	18
2	50	5	0.16	100	19	1
3	50	15	0.02	99	22	7
4	50	15	0.16	100	15	1
5	50	30	0.02	98	44	11
6	50	30	0.16	100	29	1
7	50	45	0.02	100	67	23
8	50	45	0.16	100	43	1
9	100	10	0.01	100	10	2
10	100	10	0.16	100	8	7
11	100	30	0.01	96	22	22
12	100	30	0.16	100	13	5
13	100	60	0.01	96	45	31
14	100	60	0.16	100	29	5
15	100	90	0.01	97	68	67
16	100	90	0.16	100	44	11
17	200	20	0.005	64	7	54
18	200	20	0.16	100	4	14
19	200	60	0.005	75	23	29
20	200	60	0.16	100	14	9
21	200	120	0.005	61	47	91
22	200	120	0.16	100	29	10
23	200	180	0.005	68	70	311
24	200	180	0.16	100	44	19

Table 6.23: Bark Beetle Akka results

6.4 Competitive Analysis

The graphical representation of the test results using the absolute time as a metric is shown in Figure 6.1 for the Peer Model and in Figure 6.2 for Akka. In case of 1 replica, in both the Peer Model and Akka, Bark Beetle outperforms SMP2P, Gnutella and k-Walker with much lower absolute time for all network sizes. Only for network size of 100 k-Walker has a similar absolute time to Bark Beetle. On the other hand, both AntNet and Physarum Polycephalum outperform Bark Beetle in terms of absolute time, with Physarum Polycephalum having the lowest absolute time out of all six algorithms for all network sizes.

In case of 16% replication, some discrepancies between the Peer Model and Akka occur. In the Peer Model Bark Beetle outperforms SMP2P, Gnutella, AntNet and Physarum Polycephalum for all network sizes. Only the k-Walker performs slightly better than Bark Beetle. In Akka some fluctuations occur in the performance of k-Walker. Therefore, Bark Beetle outperforms all other algorithms in terms of absolute time.
In the Peer Model Physarum Polycephalum outperforms SMP2P, Gnutella and AntNet for all network sizes. On the other hand, Bark Beetle and k-Walker outperform Physarum Polycephalum for all network sizes. In Akka Physarum Polycephalum performs better than SMP2P and Gnutella in terms of absolute time, but worse than Bark Beetle for all network sizes. Except for the network size of 100, it also outperforms AntNet. Finally, k-Walker fluctuates between much lower and much higher absolute time than Physarum Polycephalum.

The fact that Bark Beetle performs equal or better than k-Walker and Gnutella in terms of absolute time can be attributed to the fact that Bark Beetle can be considered an optimization of k-Walker since Bark Beetle uses the same random movement as k-Walker, if no sufficient pheromone concentration is found.
Since Physarum Polycephalum and AntNet use a similar agent based approach, their performance in terms of absolute time is comparable.
In case of 1 replica, Physarum Polycephalum and AntNet outperform Bark Beetle in terms of absolute time, whereas it is the other way around for 16% replication. This can be explained with the fact, that in both Physarum Polycephalum and AntNet the ants travel back and forth, if results are found, whereas Bark Beetle only informs the initiating node. Since more results are expected to be found for 16% replication, the traveling of the ants increases and thus Bark Beetle eventually outperforms both Physarum Polycephalum and AntNet.
The fact that both Bark Beetle and Physarum Polycephalum perform better than SMP2P in terms of absolute time is mainly because SMP2P completes all of its stages entirely, following the pattern from nature.

It is important to note that, although the six algorithms are very similar in relation to each other within the Peer Model and within Akka in terms of absolute time, all algorithms perform much better in Akka. On average the algorithms in Akka perform 27 times better than their counterparts in the Peer Model. It can be assumed that the communication between the nodes in the Java implementation of the Peer Model is significantly slower than in Akka. Further investigation is

needed to determine why the Peer Model performs so much slower compared to Akka, however, this is not in the scope of this thesis and is left for future research. Additionally, it has to be noted, that the Java implementation of the Peer Model is still under development. Therefore, it has to be observed whether this issue still remains in future releases.

The graphical representation of the test results using the average messages per node metric is shown in Figure 6.3 for the Peer Model and in Figure 6.4 for Akka. The following is true for both the Peer Model and Akka. Both Physarum Polycephalum and Bark Beetle outperform Gnutella, k-Walker and SMP2P in terms of message overhead in all cases. Additionally, the scalability issue of Gnutella can be observed, as its average message count gets much higher than the remaining algorithms from network size 100 and 16% replication. Due to a similar agent based approach, AntNet and Physarum Polycephalum have an almost identical message overhead in all cases. For all scenarios, Bark Beetle has the smallest message overhead, followed by Physarum Polycephalum and AntNet, which perform only slightly worse. On average Physarum Polycephalum produces 1.4 times more messages than Bark Beetle. This can be attributed to the fact that, compared to Bark Beetle, the ants in both Physarum Polycephalum and AntNet travel back and forth, which increases the message overhead.

As expected, the message overhead for each algorithm in the Peer Model is almost identical to its counterpart in Akka since it only depends on the used search algorithm, which determines which nodes interact with each other in which way in order to find the results. The underlying framework, whether the Peer Model or Akka, have no influence on this mechanism.

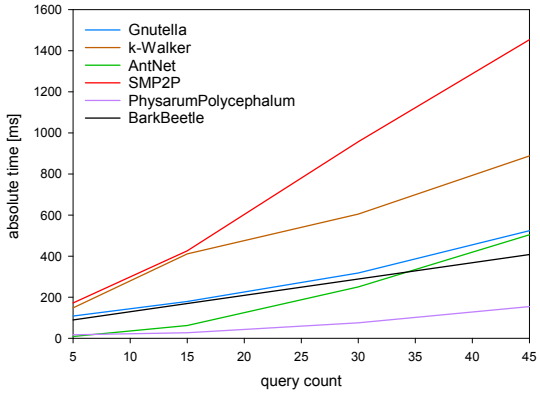
The graphical representation of the test results using the success rate metric is shown in Figure 6.5 for the Peer Model and in Figure 6.6 for Akka. Gnutella has a success rate close to 100% in almost all cases. However, as mentioned above, this comes at a cost of high message overhead.

Physarum Polycephalum and AntNet have similar success rates. In case of 1 replica, in both the Peer Model and Akka, they have the worst success rate for all network sizes with Physarum Polycephalum having a success rate between 2% and 33%.
In case of 16% replication, in both the Peer Model and Akka, they performs significantly better, with a success rate greater than or equal to the success rate of SMP2P. However, the remaining three algorithms still outperform them. Nonetheless, the success rate of Physarum Polycephalum, in case of 16% replication, varies between 80% and 98%, which can be considered as good performance.

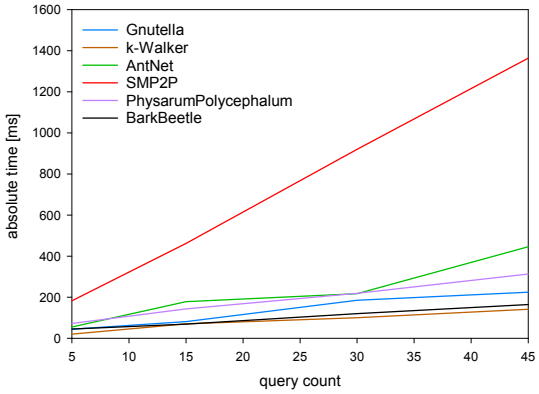
In case of 1 replica, in both the Peer Model and Akka, Bark Beetle performs better than k-Walker, AntNet, SMP2P and Physarum Polycephalum. For network sizes of 50 and 100 it even performs similarly to Gnutella with success rates between 82% and 100%. For the network size of 200 it performs worse than Gnutella with a success rate between 61% and 78%.
In case of 16% replication, in both the Peer Model and Akka, Bark Beetle has a constant success rate of 100% alongside Gnutella and k-Walker for all network sizes.

Since Bark Beetle uses the same random movement as k-Walker, if no sufficient pheromone concentration is found, the success rate of Bark Beetle is at least as good as the success rate of k-Walker. In case of 16% replication, their success rates are identical. In case of 1 replica, the additional exploitation mechanism of Bark Beetle results in a constantly better success rate for Bark Beetle compared to k-Walker.
Since Physarum Polycephalum and AntNet use a similar agent based approach, they have similar success rates.
In case of 1 replica, Physarum Polycephalum has the worst success rate. Since Bark Beetle uses a similar initial random exploration mechanism, the bad success rate of Physarum Polycephalum must be due to its exploitation mechanism. An explanation could be that the flow in Physarum Polycephalum only increases on paths, which an ant has visited. In contrast to this, the pheromone concentration in Bark Beetle is also increased on unvisited paths, since pheromones are released in a specific radius from the node with the result and not only in the direction from which the beetle came from. Thus, beetles originating from unvisited nodes are more likely to find results in Bark Beetle.

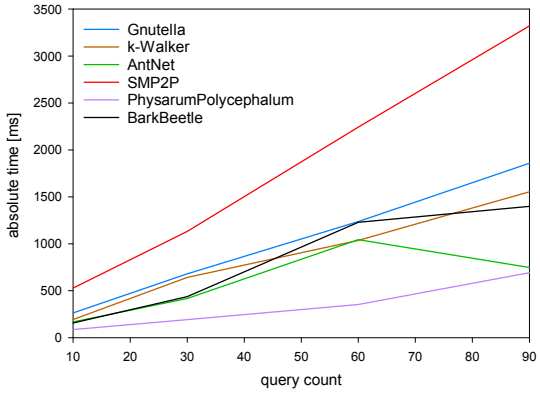
As expected, the success rate for each algorithm in the Peer Model is almost identical to its counterpart in Akka since it only depends on the used search algorithm, which determines which nodes interact with each other in which way in order to find the results. The underlying framework, whether the Peer Model or Akka, have no influence on this mechanism.



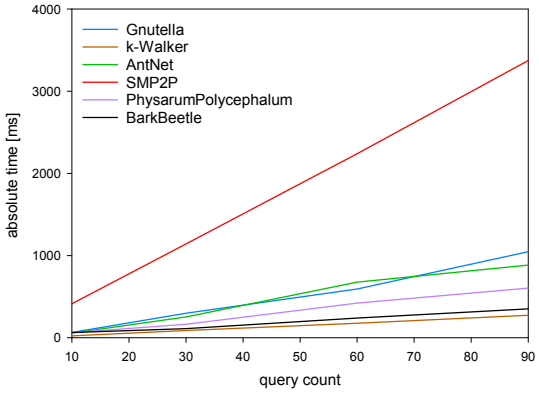
(a) 50 nodes, 2% replication



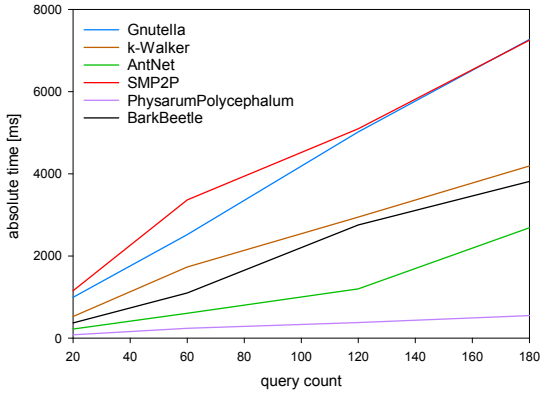
(b) 50 nodes, 16% replication



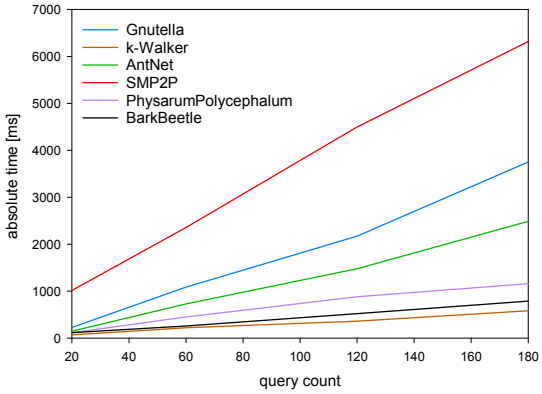
(c) 100 nodes, 1% replication



(d) 100 nodes, 16% replication

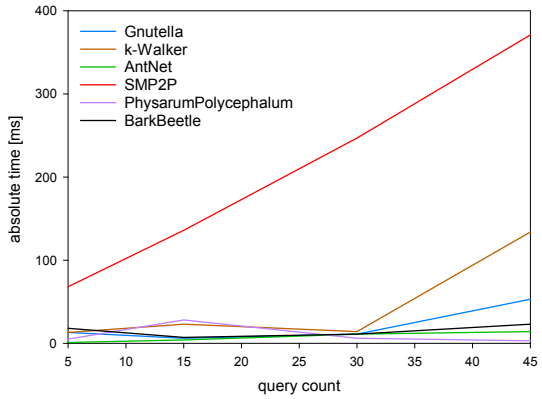


(e) 200 nodes, 0.5% replication

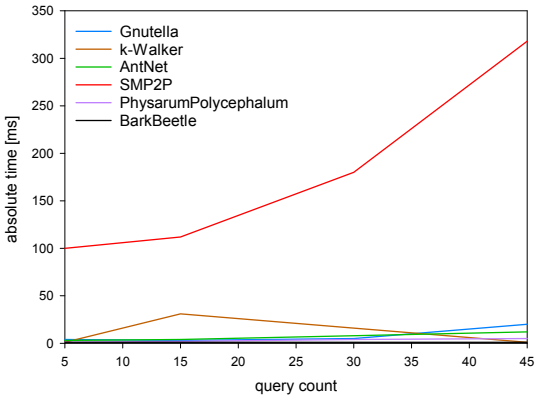


(f) 200 nodes, 16% replication

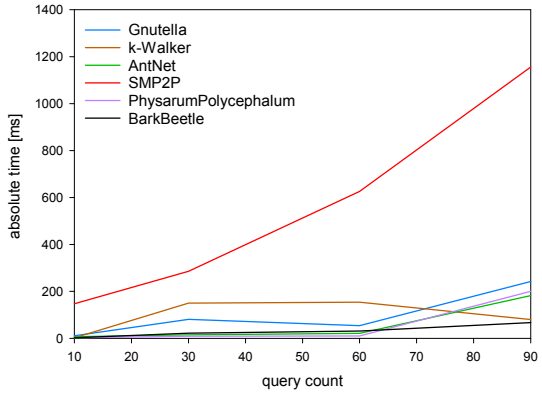
Figure 6.1: Peer Model absolute time comparison for network sizes 50, 100, 200



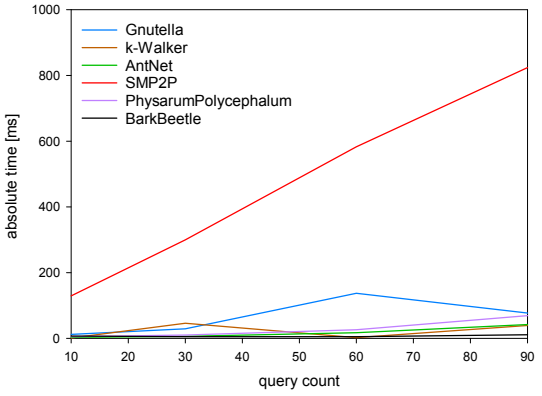
(a) 50 nodes, 2% replication



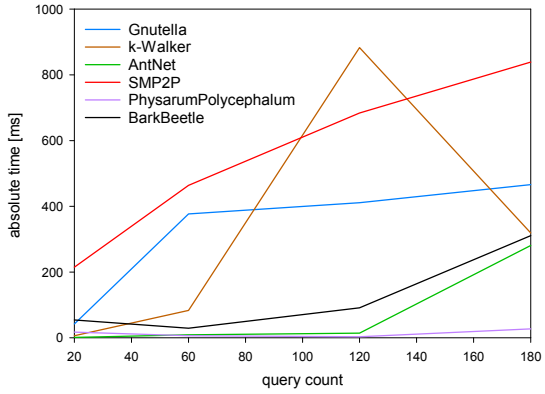
(b) 50 nodes, 16% replication



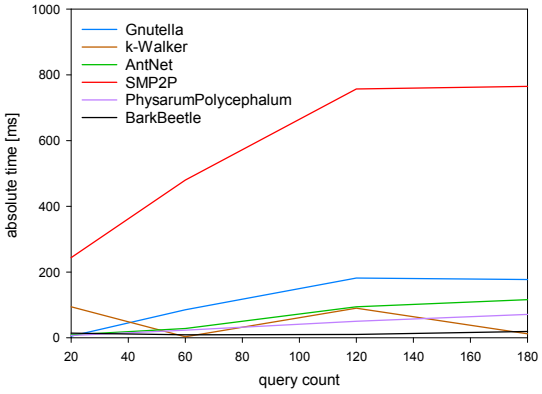
(c) 100 nodes, 1% replication



(d) 100 nodes, 16% replication

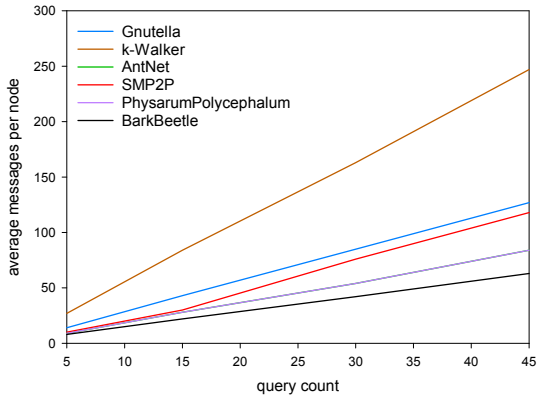


(e) 200 nodes, 0.5% replication

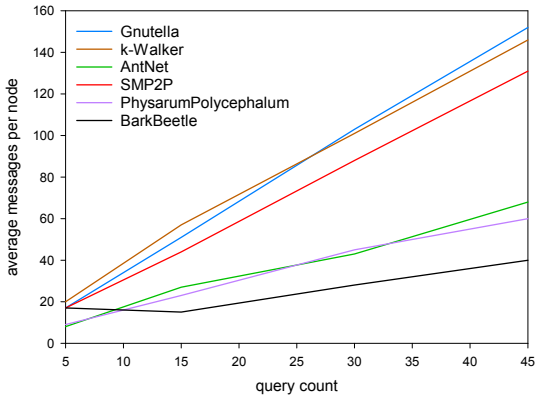


(f) 200 nodes, 16% replication

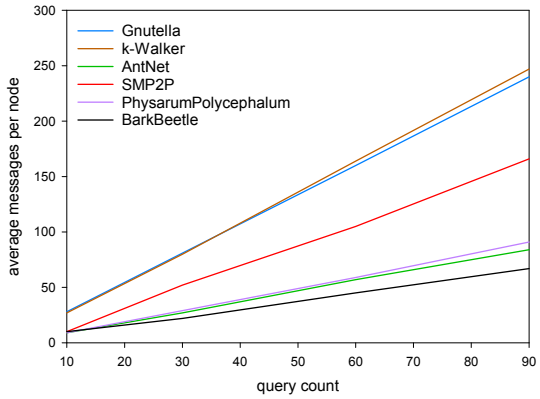
Figure 6.2: Akka absolute time comparison for network sizes 50, 100, 200



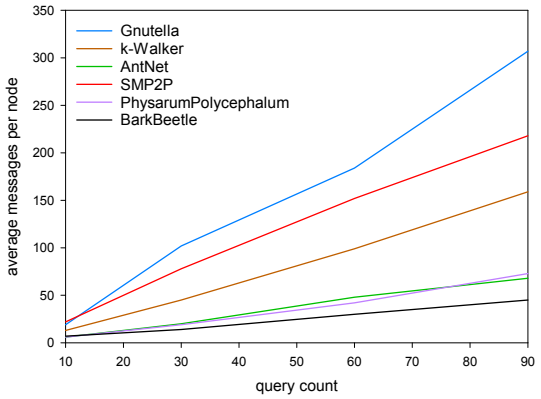
(a) 50 nodes, 2% replication



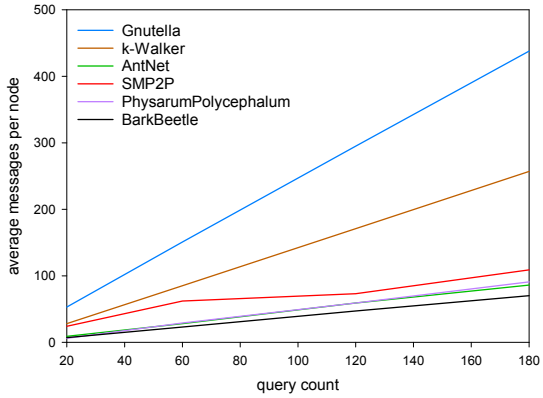
(b) 50 nodes, 16% replication



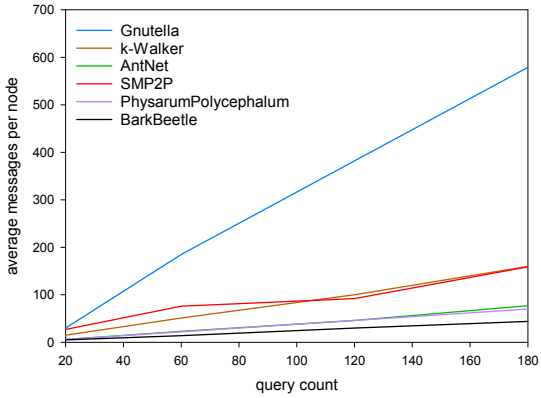
(c) 100 nodes, 1% replication



(d) 100 nodes, 16% replication

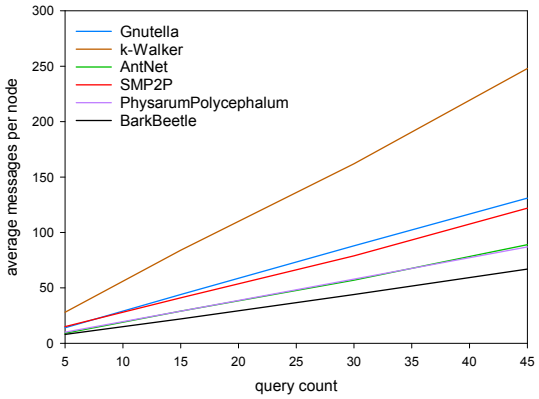


(e) 200 nodes, 0.5% replication

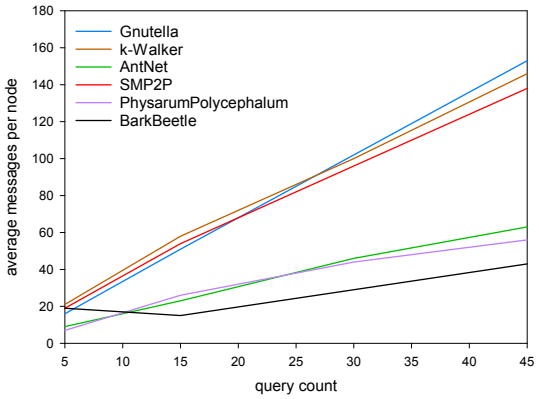


(f) 200 nodes, 16% replication

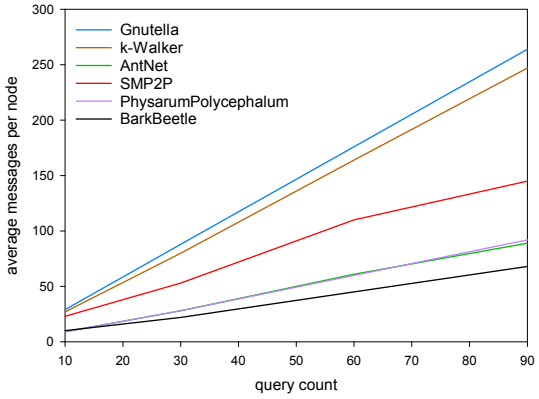
Figure 6.3: Peer Model average messages per node comparison for network sizes 50, 100, 200



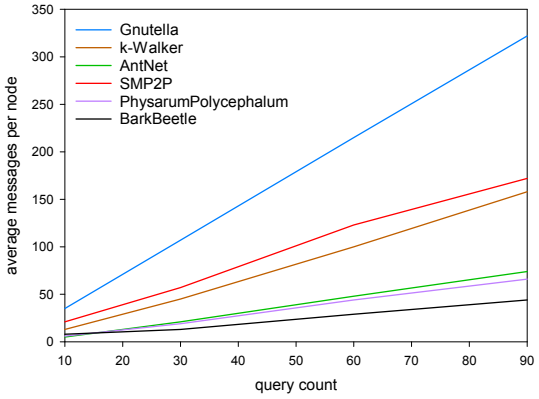
(a) 50 nodes, 2% replication



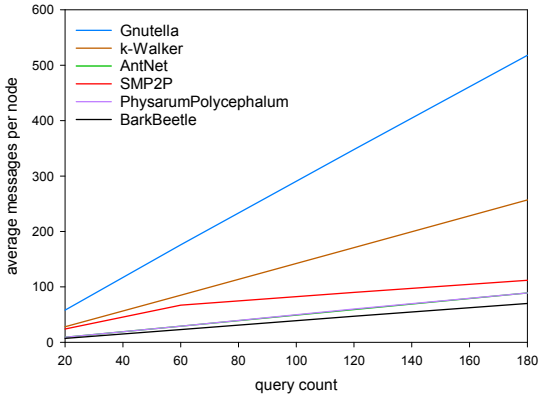
(b) 50 nodes, 16% replication



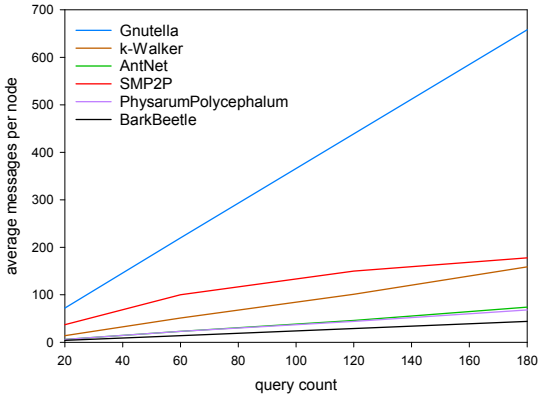
(c) 100 nodes, 1% replication



(d) 100 nodes, 16% replication

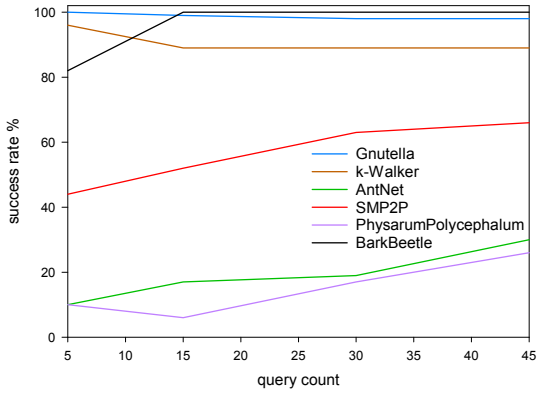


(e) 200 nodes, 0.5% replication

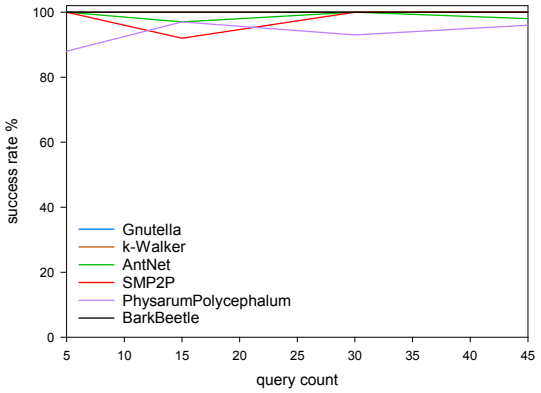


(f) 200 nodes, 16% replication

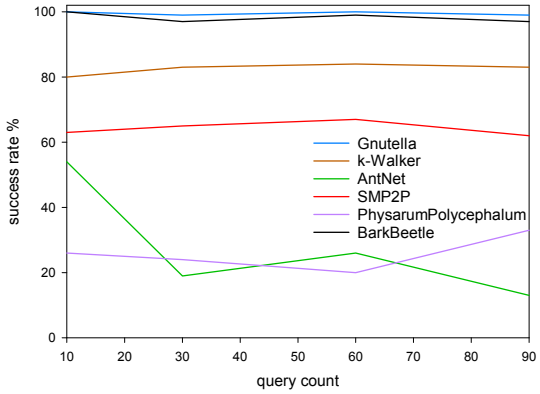
Figure 6.4: Akka average messages per node comparison for network sizes 50, 100, 200



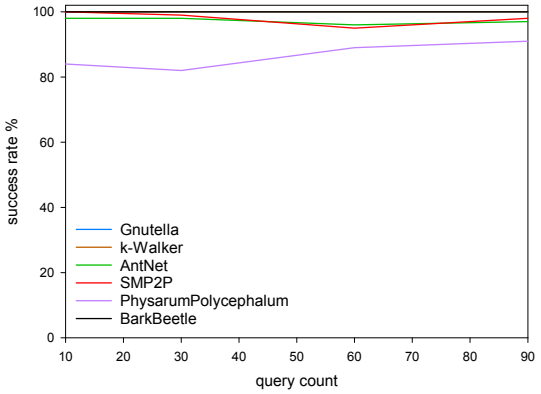
(a) 50 nodes, 2% replication



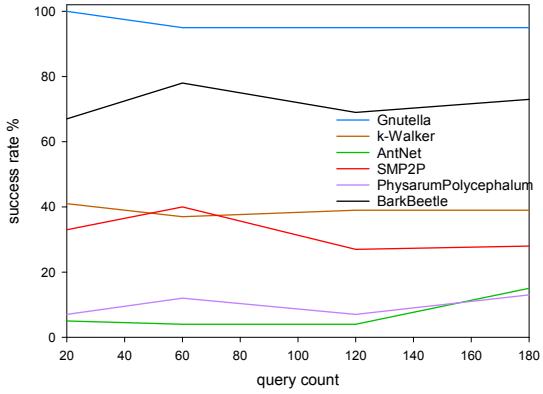
(b) 50 nodes, 16% replication



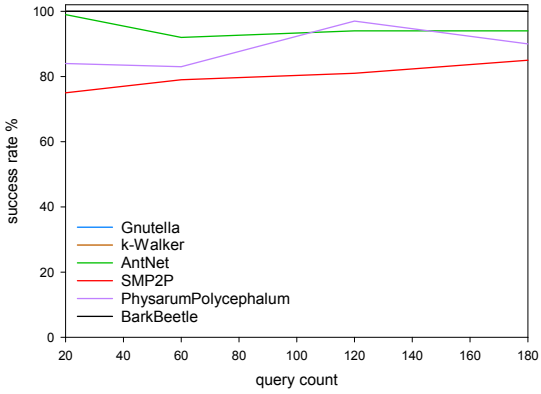
(c) 100 nodes, 1% replication



(d) 100 nodes, 16% replication

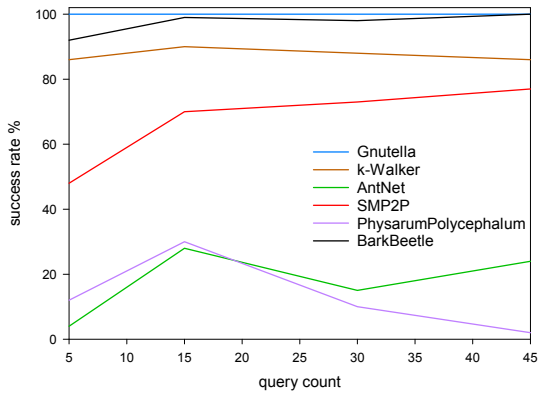


(e) 200 nodes, 0.5% replication

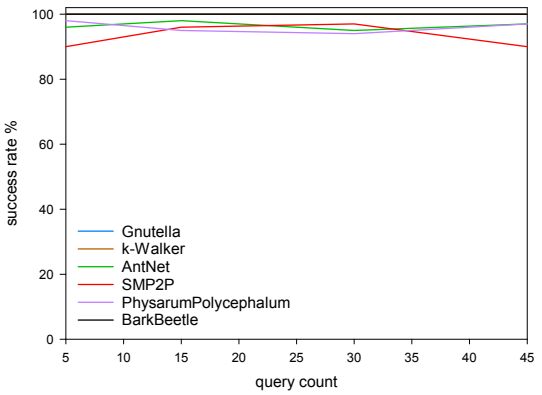


(f) 200 nodes, 16% replication

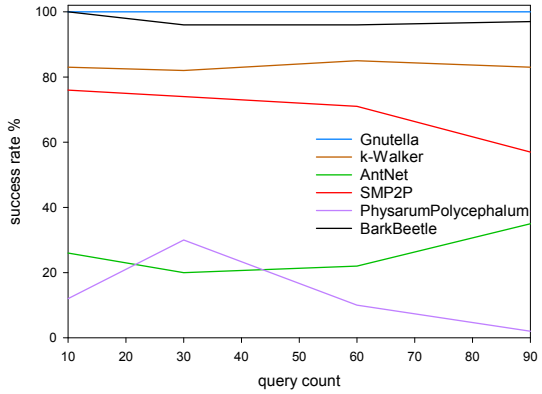
Figure 6.5: Peer Model success rate comparison for network sizes 50, 100, 200



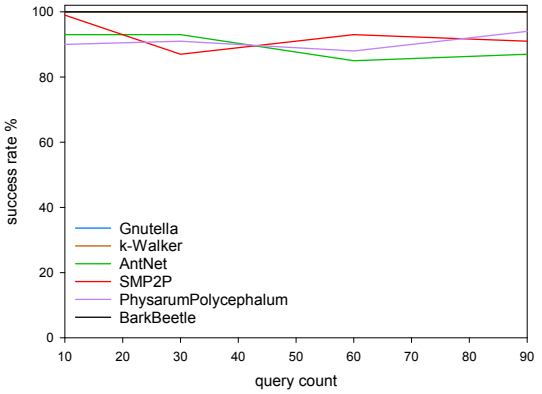
(a) 50 nodes, 2% replication



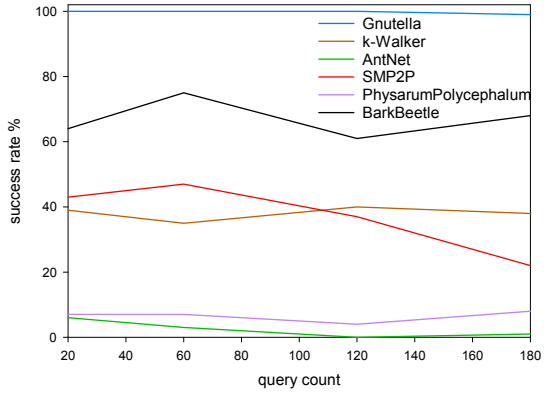
(b) 50 nodes, 16% replication



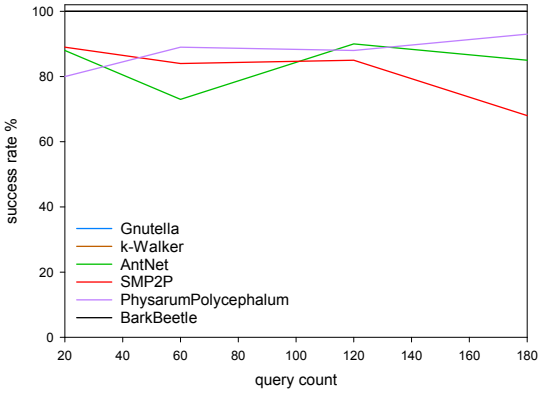
(c) 100 nodes, 1% replication



(d) 100 nodes, 16% replication



(e) 200 nodes, 0.5% replication



(f) 200 nodes, 16% replication

Figure 6.6: Akka success rate comparison for network sizes 50, 100, 200

6.5 Statistical Analysis

To compare Physarum Polycephalum Slime Mold for P2P and Bark Beetle for P2P to the remaining benchmarked search algorithms with statistical confidence an additional statistical analysis described in [27] is performed.

To this end, one-way ANOVA tests are performed and setup as follows [27]:
When two algorithms *A* and *B* are compared the null-hypothesis H_0 states that there is no significant difference between the metric value *M* of these two algorithms. H_1 is the corresponding alternative-hypothesis [27].

The tests are carried out for all different dimensions stated in Section 6.1 and the success rate is chosen as metric *M*. Additionally α = 0.05 is chosen as the significance level for the ANOVA tests [27].

First, Physarum Polycephalum Slime Mold for P2P takes the role of algorithm *A* and the remaining benchmarked algorithms including Bark Beetle for P2P take the role of algorithm *B*. The results of these tests are shown in Tables 6.24, 6.25, 6.26 and 6.27 for the Peer Model and in Tables 6.32, 6.33, 6.34 and 6.35 for Akka. Additionally, Bark Beetle for P2P takes the role of algorithm *A* and the remaining benchmarked algorithms including Physarum Polycephalum Slime Mold for P2P take the role of algorithm *B*. The results of these tests are shown in Tables 6.28, 6.29, 6.30 and 6.31 for the Peer Model and in Tables 6.36, 6.37, 6.38 and 6.39 for Akka.

For all tests the column *h* has the value 0 in the table, if H_0 is concluded. If H_0 is rejected and thus the tested algorithms are significantly different, *h* has the value 1 if algorithm *A* performs better than algorithm *B* and -1 otherwise.

Example 6.1: For the network size of 50 nodes, 5 queries and 2% replication for the Peer Model, the mean average success rate is 82 and the standard deviation is 38.24 for Bark Beetle for P2P. For the same configuration, the mean average is 10 and the standard deviation is 31.62 for AntNet.
The result of the one-way ANOVA test is p = 2.2795 10^{-04}.
Since the significance level is chosen α = 0.05 and $\alpha > p$, the null-hypothesis H_0 is rejected and the alternative-hypothesis H_1 concluded. Thus, it can be concluded that Bark Beetle for P2P performs significantly better at this configuration than AntNet. Therefore the value *h* is set to 1 in the result table.

Tables 6.24, 6.25, 6.26 and 6.27 show that for the Peer Model for network size 50 and 16% replication there is not enough data to conclude whether any other search algorithm performs better than Physarum Polycephalum. Thus, H_0 can not be rejected. For 2% replication Physarum Polycephalum performs significantly worse than the remaining algorithms with the exception of AntNet, where H_0 can not be rejected.
For the network size 100 Physarum Polycephalum performs significantly worse than the remain-

ing algorithms. Only for 1% replication not enough data is available to conclude whether AntNet performs better and thus H_0 is rejected.
For network size 200 Physarum Polycephalum performs significantly worse than the remaining algorithms with the exception of AntNet, where not enough data is available to conclude whether AntNet performs better and thus H_0 is rejected. Additionally, H_0 is rejected for SMP2P in some cases, mainly for higher loads.

Tables 6.32, 6.33, 6.34 and 6.35 show that for Akka for network size 50 and 16% replication there is not enough data to conclude whether any other search algorithm performs better than Physarum Polycephalum. Thus, H_0 can not be rejected. For 2% replication Physarum Polycephalum performs significantly worse than the remaining algorithms with the exception of AntNet, where H_0 can not be rejected.
For the network size 100 Physarum Polycephalum performs significantly worse than the remaining algorithms with the exception of AntNet, where not enough data is available to conclude whether it performs better than Physarum Polycephalum. Additionally, H_0 is rejected for SMP2P for 16% replication.
For network size 200 Physarum Polycephalum performs significantly worse than the remaining algorithms with the exception of AntNet, where not enough data is available to conclude whether it performs better than Physarum Polycephalum. Only for higher loads and 16% replication AntNet performs worse than Physarum Polycephalum. Additionally, H_0 can not be rejected for SMP2P for 16% replication.

Tables 6.28, 6.29, 6.30 and 6.31 show that for the Peer Model for network size 50 and 16% replication there is not enough data to conclude whether any other search algorithm performs better than Bark Beetle. Thus, H_0 can not be rejected. For 2% replication the remaining algorithms perform worse than Bark Beetle with the exception of Gnutella, where H_0 can not be rejected.
For network size 100 not enough data is available to conclude whether Gnutella performs better than Bark Beetle. Additionally, Physarum Polycephalum performs significantly worse than Bark Beetle. For 16% replication not enough data is available for k-Walker, AntNet and SMP2P in most cases to decide whether any of them performs better than Bark Beetle. Only for high load AntNet and SMP2P perform worse than Bark Beetle. For 1% replication k-Walker, AntNet and SMP2P perform worse than Bark Beetle.
For network size 200 AntNet, SMP2P and Physarum Polycephalum perform significantly worse than Bark Beetle. For 16% replication not enough data is available to decide whether Gnutella and k-Walker perform better than Bark Beetle. For 0.5% replication k-Walker performs significantly worse than Bark Beetle.

Tables 6.36, 6.37, 6.38 and 6.39 show that for Akka for network size 50 and 16% replication not enough data is available to decide whether any other search algorithm performs better than Bark Beetle. Thus, H_0 can not be rejected. For 2% replication Bark Beetle performs significantly better than the remaining algorithms with the exception of Gnutella, where not enough data is available to conclude whether it performs better.
For network size 100 AntNet and Physarum Polycephalum perform significantly worse than

Bark Beetle. For 16% replication not enough data is available for Gnutella, k-Walker and SMP2P in most cases to decide whether any of them performs better than Bark Beetle. Only for high load SMP2P performs worse than Bark Beetle. For 2% replication Gnutella performs slightly better than Bark Beetle and k-Walker performs significantly worse than Bark Beetle.
For network size 200 AntNet and Physarum Polycephalum perform significantly worse than Bark Beetle. For 16% replication not enough data is available for Gnutella, k-Walker and SMP2P in most cases to decide whether any of them performs better than Bark Beetle. Only for high load SMP2P performs worse than Bark Beetle. For 2% replication Gnutella performs slightly better than Bark Beetle and k-Walker and SMP2P perform significantly worse than Bark Beetle.

	mean ± stdev	p-value	h
	50 nodes, 5 queries, 2% replication		
Gnutella	100 ± 0	4.4043E-08	-1
k-Walker	96 ± 8.43	1.4213E-07	-1
AntNet	10 ± 31.62	1	0
SMP2P	44 ± 33.73	0.03194007	-1
PhysarumPolycephalum	10 ± 31.62	-	-
BarkBeetle	82 ± 38.24	2.2795E-04	-1
	50 nodes, 5 queries, 16% replication		
Gnutella	99.3 ± 2.21	0.15095045	0
k-Walker	89.7 ± 7.26	0.15095045	0
AntNet	17.2 ± 36.26	0.15095045	0
SMP2P	52.3 ± 21.34	0.15095045	0
PhysarumPolycephalum	6.6 ±20.87	-	-
BarkBeetle	100 ± 0	0.15095045	0
	50 nodes, 15 queries, 2% replication		
Gnutella	100 ± 0	4.2269E-11	1
k-Walker	100 ± 0	5.8336E-10	-1
AntNet	100 ± 0	0.43347491	0
SMP2P	100 ± 0	1.3095E-04	-1
PhysarumPolycephalum	88 ± 25.30	-	-
BarkBeetle	100 ± 0	3.4011E-11	-1
	50 nodes, 15 queries, 16% replication		
Gnutella	100 ± 0	0.20311878	0
k-Walker	100 ± 0	0.20311878	0
AntNet	97.9 ± 4.72	0.81535969	0
SMP2P	92 ± 16.87	0.36572434	0
PhysarumPolycephalum	97.5 ± 6.46	-	-
BarkBeetle	100 ± 0	0.20311878	0
	50 nodes, 30 queries, 2% replication		
Gnutella	98.8 ± 1.93	2.9802E-07	-1
k-Walker	89.8 ± 5.67	1.7899E-06	-1
AntNet	19.6 ± 30.98	0.87872408	0
SMP2P	63.7 ± 18.73	0.00105588	-1
PhysarumPolycephalum	17.4 ± 32.57	-	-
BarkBeetle	100 ± 0	2.3610E-07	-1
	50 nodes, 30 queries, 16% replication		
Gnutella	100 ± 0	0.21934783	0
k-Walker	100 ± 0	0.21934783	0
AntNet	100 ± 0	0.21934783	0
SMP2P	100 ± 0	0.21934783	0
PhysarumPolycephalum	93.2 ± 16.90	-	-
BarkBeetle	100 ± 0	0.21934783	0

Table 6.24: Physarum Polycephalum ANOVA results for the Peer Model. (part 1)

	mean ± stdev	p-value	h
	50 nodes, 45 queries, 2% replication		
Gnutella	98.9 ± 1.85	2.8142E-05	-1
k-Walker	89.8 ± 5.55	1.3647E-04	-1
AntNet	30.2 ± 38.90	0.83443840	0
SMP2P	66.4 ± 15.56	0.01013412	-1
PhysarumPolycephalum	26.4 ± 41.20	-	-
BarkBeetle	100 ± 0	2.3298E-05	-1
	50 nodes, 45 queries, 16% replication		
Gnutella	100 ± 0	0.14082450	0
k-Walker	100 ± 0	0.14082450	0
AntNet	98.9 ± 1.85	0.30097182	0
SMP2P	100 ± 0	0.14082450	0
PhysarumPolycephalum	96.2 ± 7.80	-	-
BarkBeetle	100 ± 0	0.14082450	0
	100 nodes, 10 queries, 1% replication		
Gnutella	100 ± 0	1.0125E-06	-1
k-Walker	80 ± 14.14	1.3371E-04	-1
AntNet	54 ± 38.06	0.09337904	0
SMP2P	63 ± 21.11	0.00725237	-1
PhysarumPolycephalum	26 ± 32.39	-	-
BarkBeetle	100 ± 0	1.0125E-06	-1
	100 nodes, 10 queries, 16% replication		
Gnutella	100 ± 0	0.02158285	-1
k-Walker	100 ± 0	0.02158285	-1
AntNet	98 ± 4.22	0.04499000	-1
SMP2P	100 ± 0	0.02158285	-1
PhysarumPolycephalum	84 ± 20.11	-	-
BarkBeetle	100 ± 0	0.02158285	-1
	100 nodes, 30 queries, 1% replication		
Gnutella	99.6 ± 1.26	2.5415E-07	-1
k-Walker	83.7 ± 5.14	7.2651E-06	-1
AntNet	19.4 ± 15.86	0.67272763	0
SMP2P	65.4 ± 17.56	0.00139508	-1
PhysarumPolycephalum	24 ± 29.93	-	-
BarkBeetle	100 ± 0	2.3277E-07	-1
	100 nodes, 30 queries, 16% replication		
Gnutella	100 ± 0	0.01616182	-1
k-Walker	100 ± 0	0.01616182	-1
AntNet	96.8 ± 1.93	0.02395266	-1
SMP2P	99.6 ± 1.26	0.01848935	-1
PhysarumPolycephalum	82.1 ± 21.33	-	-
BarkBeetle	100 ± 0	0.01616182	-1

Table 6.25: Physarum Polycephalum ANOVA results for the Peer Model. (part 2)

	mean ± stdev	p-value	h
	100 nodes, 60 queries, 1% replication		
Gnutella	100 ± 0	5.8881E-08	-1
k-Walker	84.5 ± 3.60	1.4274E-06	-1
AntNet	26.6 ± 28.97	0.65118517	0
SMP2P	67.5 ± 9.07	1.0084E-04	-1
PhysarumPolycephalum	20.7 ± 28.41	-	-
BarkBeetle	99.6 ± 0.84	6.3862E-08	-1
	100 nodes, 60 queries, 16% replication		
Gnutella	100 ± 0	0.00645356	-1
k-Walker	100 ± 0	0.00645356	-1
AntNet	96.2 ± 5.20	0.08944444	0
SMP2P	95.9 ± 11.59	0.20766610	0
PhysarumPolycephalum	89.3 ± 10.99	-	-
BarkBeetle	100 ± 0	0.00645356	-1
	100 nodes, 90 queries, 1% replication		
Gnutella	99.8 ± 0.63	7.5532E-06	-1
k-Walker	83.9 ± 2.42	1.7783E-04	-1
AntNet	13 ± 21.36	0.12466996	0
SMP2P	62.1 ± 9.19	0.01866677	-1
PhysarumPolycephalum	33.4 ± 33.88	-	-
BarkBeetle	94.4 ± 1.51	1.2039E-05	-1
	100 nodes, 90 queries, 16% replication		
Gnutella	100 ± 0	0.00583697	-1
k-Walker	100 ± 0	0.00583697	-1
AntNet	97.5 ± 3.24	0.04750452	-1
SMP2P	98.1 ± 1.73	0.02617098	-1
PhysarumPolycephalum	91 ± 9.10	-	-
BarkBeetle	100 ± 0	0.00583697	-1
	200 nodes, 20 queries, 0.5% replication		
Gnutella	100 ± 0	1.0246E-13	-1
k-Walker	41.5 ± 11.56	1.6327E-05	-1
AntNet	5 ± 10.80	0.73347642	0
SMP2P	33.5 ± 12.92	4.5830E-04	-1
PhysarumPolycephalum	7 ± 14.76	-	-
BarkBeetle	67.5 ± 18.60	2.2080E-07	-1
	200 nodes, 20 queries, 16% replication		
Gnutella	100 ± 0	0.00848909	-1
k-Walker	100 ± 0	0.00848909	-1
AntNet	99 ± 2.11	0.01320576	-1
SMP2P	75 ± 30.00	0.42079324	-1
PhysarumPolycephalum	84 ± 17.13	-	-
BarkBeetle	100 ± 0	0.00848909	-1

Table 6.26: Physarum Polycephalum ANOVA results for the Peer Model. (part 3)

	mean ± stdev	p-value	h
	200 nodes, 60 queries, 0.5% replication		
Gnutella	95.7 ± 1.25	2.9967E-08	-1
k-Walker	37.5 ± 5.84	0.01395918	-1
AntNet	4.1 ± 8.18	0.36957880	0
SMP2P	40.4 ± 9.07	0.00877951	-1
PhysarumPolycephalum	12.7 ± 28.39	-	-
BarkBeetle	78.3 ± 6.04	1.1762E-06	-1
	200 nodes, 60 queries, 16% replication		
Gnutella	100 ± 0	0.00125692	-1
k-Walker	100 ± 0	0.00125692	-1
AntNet	92 ± 8.78	0.11275284	0
SMP2P	79.7 ± 30.96	0.73380789	0
PhysarumPolycephalum	83.4 ± 13.75	-	-
BarkBeetle	100 ± 0	0.00125692	-1
	200 nodes, 120 queries, 0.5% replication		
Gnutella	95.7 ± 2.16	1.1454E-13	-1
k-Walker	39.6 ± 5.95	2.9840E-06	-1
AntNet	4.6 ± 13.86	0.61206699	0
SMP2P	27.5 ± 24.60	0.04062914	-1
PhysarumPolycephalum	7.8 ± 13.87	-	-
BarkBeetle	69.8 ± 6.61	1.8688E-10	-1
	200 nodes, 120 queries, 16% replication		
Gnutella	100 ± 0	0.00519785	-1
k-Walker	100 ± 0	0.00519785	-1
AntNet	94.3 ± 6.22	0.21285441	0
SMP2P	81.6 ± 27.11	0.08898985	0
PhysarumPolycephalum	97.1 ± 2.88	-	-
BarkBeetle	100 ± 0	0.00519785	-1
	200 nodes, 180 queries, 0.5% replication		
Gnutella	95.5 ± 1.72	5.5228E-08	-1
k-Walker	39.5 ± 3.24	0.01246205	-1
AntNet	15.7 ± 22.11	0.87122999	0
SMP2P	28.8 ± 20.68	0.20049015	0
PhysarumPolycephalum	13.8 ± 29.10	-	-
BarkBeetle	73.9 ± 4.25	4.4279E-06	-1
	200 nodes, 180 queries, 16% replication		
Gnutella	100 ± 0	0.00248945	-1
k-Walker	100 ± 0	0.00248945	-1
AntNet	94.3 ± 4.76	0.26186723	0
SMP2P	85.3 ± 17.41	0.37893229	0
PhysarumPolycephalum	90.8 ± 8.28	-	-
BarkBeetle	100 ± 0	0.00248945	-1

Table 6.27: Physarum Polycephalum ANOVA results for the Peer Model. (part 4)

	mean ± stdev	p-value	h
	50 nodes, 5 queries, 2% replication		
Gnutella	100 ± 0	0.15391495	0
k-Walker	96 ± 8.43	0.27306288	0
AntNet	10 ± 31.62	2.2795E-04	1
SMP2P	44 ± 33.73	0.02997064	1
PhysarumPolycephalum	10 ± 31.62	2.2795E-04	1
BarkBeetle	82 ± 38.24	-	-
	50 nodes, 5 queries, 16% replication		
Gnutella	100 ± 0	-	0
k-Walker	100 ± 0	-	0
AntNet	100 ± 0	-	0
SMP2P	100 ± 0	-	0
PhysarumPolycephalum	88 ± 25.30	0.15095045	0
BarkBeetle	100 ± 0	-	-
	50 nodes, 15 queries, 2% replication		
Gnutella	99.3 ± 2.21	0.33056493	0
k-Walker	89.7 ± 7.26	2.8462E-04	1
AntNet	17.2 ± 36.26	1.0212E-06	1
SMP2P	52.3 ± 21.34	1.3622E-06	1
PhysarumPolycephalum	6.6 ±20.87	3.4011E-11	1
BarkBeetle	100 ± 0	-	-
	50 nodes, 15 queries, 16% replication		
Gnutella	100 ± 0	-	0
k-Walker	100 ± 0	-	0
AntNet	97.9 ± 4.72	0.17687965	0
SMP2P	92 ± 16.87	0.15095045	0
PhysarumPolycephalum	97.5 ± 6.46	0.20311878	0
BarkBeetle	100 ± 0	-	-
	50 nodes, 30 queries, 2% replication		
Gnutella	98.8 ± 1.93	0.06516949	0
k-Walker	89.8 ± 5.67	2.1565E-05	1
AntNet	19.6 ± 30.98	1.7042E-07	1
SMP2P	63.7 ± 18.73	8.6813E-06	1
PhysarumPolycephalum	17.4 ± 32.57	2.3610E-07	1
BarkBeetle	100 ± 0	-	-
	50 nodes, 30 queries, 16% replication		
Gnutella	100 ± 0	-	0
k-Walker	100 ± 0	-	0
AntNet	100 ± 0	-	0
SMP2P	100 ± 0	-	0
PhysarumPolycephalum	93.2 ± 16.90	0.21934783	0
BarkBeetle	100 ± 0	-	-

Table 6.28: Bark Beetle ANOVA results for the Peer Model. (part 1)

	mean ± stdev	p-value	h
	50 nodes, 45 queries, 2% replication		
Gnutella	98.9 ± 1.85	0.07678035	0
k-Walker	89.8 ± 5.55	1.6746E-05	1
AntNet	30.2 ± 38.90	2.2138E-05	1
SMP2P	66.4 ± 15.56	2.1571E-06	1
PhysarumPolycephalum	26.4 ± 41.20	2.3298E-05	1
BarkBeetle	100 ± 0	-	-
	50 nodes, 45 queries, 16% replication		
Gnutella	100 ± 0	-	0
k-Walker	100 ± 0	-	0
AntNet	98.9 ± 1.85	0.07678035	0
SMP2P	100 ± 0	-	0
PhysarumPolycephalum	96.2 ± 7.80	0.14082450	0
BarkBeetle	100 ± 0	-	-
	100 nodes, 10 queries, 1% replication		
Gnutella	100 ± 0	-	0
k-Walker	80 ± 14.14	2.9456E-04	1
AntNet	54 ± 38.06	0.00124988	1
SMP2P	63 ± 21.11	2.9111E-05	1
PhysarumPolycephalum	26 ± 32.39	1.1012E-06	1
BarkBeetle	100 ± 0	-	-
	100 nodes, 10 queries, 16% replication		
Gnutella	100 ± 0	-	0
k-Walker	100 ± 0	-	0
AntNet	98 ± 4.22	0.15095045	0
SMP2P	100 ± 0	-	0
PhysarumPolycephalum	84 ± 20.11	0.02158285	1
BarkBeetle	100 ± 0	-	-
	100 nodes, 30 queries, 1% replication		
Gnutella	99.6 ± 1.26	0.33056493	0
k-Walker	83.7 ± 5.14	8.6416E-09	1
AntNet	19.4 ± 15.86	4.0534E-12	1
SMP2P	65.4 ± 17.56	7.0672E-06	1
PhysarumPolycephalum	24 ± 29.93	2.3277E-07	1
BarkBeetle	100 ± 0	-	-
	100 nodes, 30 queries, 16% replication		
Gnutella	100 ± 0	-	0
k-Walker	100 ± 0	-	0
AntNet	96.8 ± 1.93	0.06516949	0
SMP2P	99.6 ± 1.26	0.33056493	0
PhysarumPolycephalum	82.1 ± 21.33	0.01616182	1
BarkBeetle	100 ± 0	-	-

Table 6.29: Bark Beetle ANOVA results for the Peer Model. (part 2)

	mean ± stdev	p-value	h
	100 nodes, 60 queries, 1% replication		
Gnutella	100 ± 0	0.15095045	0
k-Walker	84.5 ± 3.60	1.5194E-10	1
AntNet	26.6 ± 28.97	2.6107E-07	1
SMP2P	67.5 ± 9.07	1.6480E-09	1
PhysarumPolycephalum	20.7 ± 28.41	6.3862E-08	1
BarkBeetle	99.6 ± 0.84	-	-
	100 nodes, 60 queries, 16% replication		
Gnutella	100 ± 0	-	0
k-Walker	100 ± 0	-	0
AntNet	96.2 ± 5.20	0.03297016	1
SMP2P	95.9 ± 11.59	0.27797989	0
PhysarumPolycephalum	89.3 ± 10.99	0.00645356	1
BarkBeetle	100 ± 0	-	-
	100 nodes, 90 queries, 1% replication		
Gnutella	99.8 ± 0.63	2.0019E-04	-1
k-Walker	83.9 ± 2.42	1.3494E-11	1
AntNet	13 ± 21.36	2.7421E-10	1
SMP2P	62.1 ± 9.19	5.1780E-10	1
PhysarumPolycephalum	33.4 ± 33.88	1.2039E-05	1
BarkBeetle	94.4 ± 1.51	-	-
	100 nodes, 90 queries, 16% replication		
Gnutella	100 ± 0	-	0
k-Walker	100 ± 0	-	0
AntNet	97.5 ± 3.24	0.02527264	1
SMP2P	98.1 ± 1.73	0.00270051	1
PhysarumPolycephalum	91 ± 9.10	0.00583697	1
BarkBeetle	100 ± 0	-	-
	200 nodes, 20 queries, 0.5% replication		
Gnutella	100 ± 0	3.0145E-05	1
k-Walker	41.5 ± 11.56	0.00144986	1
AntNet	5 ± 10.80	3.225E-08	1
SMP2P	33.5 ± 12.92	1.6064E-04	1
PhysarumPolycephalum	7 ± 14.76	2.2080E-07	1
BarkBeetle	67.5 ± 18.60	-	-
	200 nodes, 20 queries, 16% replication		
Gnutella	100 ± 0	-	0
k-Walker	100 ± 0	-	0
AntNet	99 ± 2.11	0.15095045	0
SMP2P	75 ± 30.00	0.01680490	1
PhysarumPolycephalum	84 ± 17.13	0.00848909	1
BarkBeetle	100 ± 0	-	-

Table 6.30: Bark Beetle ANOVA results for the Peer Model. (part 3)

	mean ± stdev	p-value	h
	200 nodes, 60 queries, 0.5% replication		
Gnutella	95.7 ± 1.25	5.0010E-08	-1
k-Walker	37.5 ± 5.84	2.1587E-12	1
AntNet	4.1 ± 8.18	8.0781E-15	1
SMP2P	40.4 ± 9.07	2.0223E-09	1
PhysarumPolycephalum	12.7 ± 28.39	1.1762E-06	1
BarkBeetle	78.3 ± 6.04	-	-
	200 nodes, 60 queries, 16% replication		
Gnutella	100 ± 0	-	0
k-Walker	100 ± 0	-	0
AntNet	92 ± 8.78	0.00994663	1
SMP2P	79.7 ± 30.96	0.05277479	0
PhysarumPolycephalum	83.4 ± 13.75	0.00125692	1
BarkBeetle	100 ± 0	-	-
	200 nodes, 120 queries, 0.5% replication		
Gnutella	95.7 ± 2.16	6.9740E-10	-1
k-Walker	39.6 ± 5.95	2.9569E-09	1
AntNet	4.6 ± 13.86	8.0979E-11	1
SMP2P	27.5 ± 24.60	5.4088E-05	1
PhysarumPolycephalum	7.8 ± 13.87	1.8688E-10	1
BarkBeetle	69.8 ± 6.61	-	-
	200 nodes, 120 queries, 16% replication		
Gnutella	100 ± 0	-	0
k-Walker	100 ± 0	-	0
AntNet	94.3 ± 6.22	0.00958040	1
SMP2P	81.6 ± 27.11	0.04572846	1
PhysarumPolycephalum	97.1 ± 2.88	0.00519785	1
BarkBeetle	100 ± 0	-	-
	200 nodes, 180 queries, 0.5% replication		
Gnutella	95.5 ± 1.72	1.4584E-11	-1
k-Walker	39.5 ± 3.24	7.1917E-14	1
AntNet	15.7 ± 22.11	1.7967E-07	1
SMP2P	28.8 ± 20.68	2.4877E-06	1
PhysarumPolycephalum	13.8 ± 29.10	4.4279E-06	1
BarkBeetle	73.9 ± 4.25	-	-
	200 nodes, 180 queries, 16% replication		
Gnutella	100 ± 0	-	0
k-Walker	100 ± 0	-	0
AntNet	94.3 ± 4.76	0.00135574	1
SMP2P	85.3 ± 17.41	0.01561466	1
PhysarumPolycephalum	90.8 ± 8.28	0.00248945	1
BarkBeetle	100 ± 0	-	-

Table 6.31: Bark Beetle ANOVA results for the Peer Model. (part 4)

	mean ± stdev	p-value	h
	50 nodes, 5 queries, 2% replication		
Gnutella	100 ± 0	2.0208E-09	-1
k-Walker	86 ± 16.47	3.8224E-07	-1
AntNet	4 ± 12.65	0.38289757	0
SMP2P	48 ± 32.93	0.01341518	-1
PhysarumPolycephalum	12 ± 25.30	-	-
BarkBeetle	92 ± 13.98	6.6641E-08	-1
	50 nodes, 5 queries, 16% replication		
Gnutella	100 ± 0	0.33056493	0
k-Walker	100 ± 0	0.33056493	0
AntNet	96 ± 8.43	0.55598517	0
SMP2P	90 ± 25.39	0.34636414	0
PhysarumPolycephalum	98 ± 6.32	-	-
BarkBeetle	100 ± 0	0.33056493	0
	50 nodes, 15 queries, 2% replication		
Gnutella	100 ± 0	3.6312E-05	-1
k-Walker	90.9 ± 4.72	1.8068E-04	-1
AntNet	28.6 ± 46.09	0.92300087	0
SMP2P	70.4 ± 17.27	0.01016171	-1
PhysarumPolycephalum	30.5 ± 40.41	-	-
BarkBeetle	99.3 ± 2.21	4.1488E-05	-1
	50 nodes, 15 queries, 16% replication		
Gnutella	100 ± 0	0.08993915	0
k-Walker	100 ± 0	0.08993915	0
AntNet	98.6 ± 4.43	0.27535471	0
SMP2P	96.5 ± 4.95	0.68012756	0
PhysarumPolycephalum	95.2 ± 8.47	-	-
BarkBeetle	100 ± 0	0.08993915	0
	50 nodes, 30 queries, 2% replication		
Gnutella	100 ± 0	8.7240E-10	-1
k-Walker	88.7 ± 6.63	1.2689E-08	-1
AntNet	15.9 ± 29.22	0.67062207	0
SMP2P	73.3 ± 11.45	7.9129E-07	-1
PhysarumPolycephalum	10.7 ± 24.35	-	-
BarkBeetle	98.9 ± 2.42	1.1501E-09	-1
	50 nodes, 30 queries, 16% replication		
Gnutella	100 ± 0	0.15161740	0
k-Walker	100 ± 0	0.15161740	0
AntNet	95.6 ± 9.45	0.79130501	0
SMP2P	97.2 ± 6.27	0.50780807	0
PhysarumPolycephalum	94.3 ± 12.04	-	-
BarkBeetle	100 ± 0	0.15161740	0

Table 6.32: Physarum Polycephalum ANOVA results for Akka. (part 1)

	mean ± stdev	p-value	h
	50 nodes, 45 queries, 2% replication		
Gnutella	100 ± 0	-	0
k-Walker	86.6 ± 4.55	5.0073E-22	-1
AntNet	24 ± 30.20	0.03335847	-1
SMP2P	77.4 ± 7.79	5.6462E-17	-1
PhysarumPolycephalum	2 ± 0	-	-
BarkBeetle	100 ± 0	-	0
	50 nodes, 45 queries, 16% replication		
Gnutella	100 ± 0	0.08901177	0
k-Walker	100 ± 0	0.08901177	0
AntNet	97.8 ± 4.49	0.91941653	0
SMP2P	90.9 ± 19.30	0.29772822	0
PhysarumPolycephalum	97.6 ± 4.22	-	-
BarkBeetle	100 ± 0	0.08901177	0
	100 nodes, 10 queries, 1% replication		
Gnutella	100 ± 0	1.2314E-12	-1
k-Walker	83 ± 9.49	6.1543E-10	-1
AntNet	26 ± 23.66	0.19875772	0
SMP2P	76 ± 21.71	8.6912E-07	-1
PhysarumPolycephalum	14 ± 15.78	-	-
BarkBeetle	100 ± 0	1.2314E-12	-1
	100 nodes, 10 queries, 16% replication		
Gnutella	100 ± 0	0.00353274	-1
k-Walker	100 ± 0	0.00353274	-1
AntNet	93 ± 9.49	0.48722547	0
SMP2P	99 ± 3.16	0.01036076	-1
PhysarumPolycephalum	90 ± 9.43	-	-
BarkBeetle	100 ± 0	0.00353274	-1
	100 nodes, 30 queries, 1% replication		
Gnutella	100 ± 0	4.2500E-13	-1
k-Walker	82.6 ± 5.08	4.0138E-11	-1
AntNet	20.5 ± 23.10	0.29287663	0
SMP2P	74.1 ± 20.08	2.9487E-07	-1
PhysarumPolycephalum	11 ± 15.34	-	-
BarkBeetle	96.1 ± 5.63	2.6719E-12	-1
	100 nodes, 30 queries, 16% replication		
Gnutella	100 ± 0	0.01964082	-1
k-Walker	100 ± 0	0.01964082	-1
AntNet	93.2 ± 10.59	0.66865900	0
SMP2P	87.4 ± 19.02	0.60081442	0
PhysarumPolycephalum	91.1 ± 10.99	-	-
BarkBeetle	100 ±0	0.01964082	-1

Table 6.33: Physarum Polycephalum ANOVA results for Akka. (part 2)

	mean ± stdev	p-value	h
	100 nodes, 60 queries, 1% replication		
Gnutella	100 ± 0	1.1865E-05	-1
k-Walker	85.5 ± 3.66	1.5197E-04	-1
AntNet	22 ± 29.45	0.76880928	0
SMP2P	71.3 ± 10.34	0.00246612	-1
PhysarumPolycephalum	26.6 ± 28.85	-	-
BarkBeetle	96.8 ± 2.62	2.0888E-05	-1
	100 nodes, 60 queries, 16% replication		
Gnutella	100 ± 0	0.02167527	-1
k-Walker	83.9 ± 4.63	0.02167527	-1
AntNet	35.5 ± 37.89	0.66061462	0
SMP2P	57.5 ± 11.71	0.52027975	0
PhysarumPolycephalum	23.1 ± 24.50	-	-
BarkBeetle	97.8 ±1.32	0.02167527	-1
	100 nodes, 90 queries, 1% replication		
Gnutella	100 ± 0	9.9979E-09	-1
k-Walker	100 ± 0	4.1140E-07	-1
AntNet	87.7 ± 12.10	0.39624010	0
SMP2P	91.7 ±12.82	8.2808E-04	1
PhysarumPolycephalum	94.5 ± 11.78	-	-
BarkBeetle	100 ± 0	1.5928E-08	-1
	100 nodes, 90 queries, 16% replication		
Gnutella	100 ± 0	0.15704017	0
k-Walker	100 ± 0	0.15704017	0
AntNet	85 ± 19.16	0.21908986	0
SMP2P	92.1 ± 10.27	0.61713549	0
PhysarumPolycephalum	88.4 ± 14.59	-	-
BarkBeetle	100 ± 0	0.15704017	0
	200 nodes, 20 queries, 0.5% replication		
Gnutella	100 ± 0	3.4488E-15	-1
k-Walker	39.5 ± 8.96	2.6198E-06	-1
AntNet	6 ± 15.78	0.81399565	0
SMP2P	43.5 ± 26.57	0.00104725	-1
PhysarumPolycephalum	7.5 ± 12.08	-	-
BarkBeetle	64 ± 15.06	2.8889E-08	-1
	200 nodes, 20 queries, 16% replication		
Gnutella	100 ± 0	0.00109567	-1
k-Walker	100 ±0	0.00109567	-1
AntNet	88.5 ± 13.13	0.23557719	0
SMP2P	89.5 ± 21.27	0.29793374	0
PhysarumPolycephalum	80.5 ± 15.89	-	-
BarkBeetle	100 ± 0	0.00109567	-1

Table 6.34: Physarum Polycephalum ANOVA results for Akka. (part 3)

	mean ± stdev	p-value	h
	200 nodes, 60 queries, 0.5% replication		
Gnutella	100 ± 0	6.0831E-13	-1
k-Walker	35.3 ± 2.79	4.3905E-05	-1
AntNet	3.8 ± 11.00	0.58078163	0
SMP2P	47.2 ± 31.84	0.00240766	-1
PhysarumPolycephalum	7.3 ± 16.32	-	-
BarkBeetle	75.9 ± 7.55	4.6180E-10	-1
	200 nodes, 60 queries, 16% replication		
Gnutella	100 ± 0	0.00110203	-1
k-Walker	100 ± 0	0.00110203	-1
AntNet	73.6 ± 15.12	0.01019658	1
SMP2P	84.8 ± 26.19	0.60439182	0
PhysarumPolycephalum	89.4 ± 8.64	-	-
BarkBeetle	100 ± 0	0.00110203	-1
	200 nodes, 120 queries, 0.5% replication		
Gnutella	100 ± 0	9.4876E-15	-1
k-Walker	40.8 ± 5.12	1.9405E-07	-1
AntNet	0.9 ± 2.85	0.45228718	0
SMP2P	37.6 ± 20.23	3.7346E-04	-1
PhysarumPolycephalum	4.2 ± 13.28	-	-
BarkBeetle	61.7 ± 9.84	2.0241E-09	-1
	200 nodes, 120 queries, 16% replication		
Gnutella	100 ± 0	0.00888506	-1
k-Walker	100 ± 0	0.00888506	-1
AntNet	90.6 ± 9.59	0.73003719	0
SMP2P	85.7 ± 23.41	0.70482597	0
PhysarumPolycephalum	88.9 ±11.97	-	-
BarkBeetle	100 ±0	0.00888506	-1
	200 nodes, 180 queries, 0.5% replication		
Gnutella	99.8 ± 0.42	1.1949E-10	-1
k-Walker	38.7 ± 2.54	4.0423E-04	-1
AntNet	1.1 ± 3.14	0.32043194	0
SMP2P	22.4 ± 12.41	0.09512890	0
PhysarumPolycephalum	8.3 ± 22.06	-	-
BarkBeetle	68.7 ± 6.40	1.4069E-07	-1
	200 nodes, 180 queries, 16% replication		
Gnutella	100 ± 0	4.5163E-04	-1
k-Walker	100 ± 0	4.5163E-04	-1
AntNet	85.9 ± 8.96	0.02132247	1
SMP2P	68.6 ± 33.22	0.02821326	1
PhysarumPolycephalum	93.9 ± 4.51	-	-
BarkBeetle	100 ± 0	4.5163E-04	-1

Table 6.35: Physarum Polycephalum ANOVA results for Akka (part 4)

	mean ± stdev	p-value	h
	50 nodes, 5 queries, 2% replication		
Gnutella	100 ± 0	0.08717413	0
k-Walker	86 ± 16.47	0.39134916	0
AntNet	4 ± 12.65	1.6916E-11	1
SMP2P	48 ± 32.93	0.00107525	1
PhysarumPolycephalum	12 ± 25.30	6.6641E-08	1
BarkBeetle	92 ± 13.98	-	-
	50 nodes, 5 queries, 16% replication		
Gnutella	100 ± 0	-	0
k-Walker	100 ± 0	-	0
AntNet	96 ± 8.43	0.15095045	0
SMP2P	90 ± 25.39	0.22885251	0
PhysarumPolycephalum	98 ± 6.32	0.33056493	0
BarkBeetle	100 ± 0	-	-
	50 nodes, 15 queries, 2% replication		
Gnutella	100 ± 0	0.33056493	0
k-Walker	90.9 ± 4.72	7.6248E-05	1
AntNet	28.6 ± 46.09	1.2993E-04	1
SMP2P	70.4 ± 17.27	5.4361E-05	1
PhysarumPolycephalum	30.5 ± 40.41	4.1488E-05	1
BarkBeetle	99.3 ± 2.21	-	-
	50 nodes, 15 queries, 16% replication		
Gnutella	100 ± 0	-	0
k-Walker	100 ± 0	-	0
AntNet	98.6 ± 4.43	0.33056493	0
SMP2P	96.5 ± 4.95	0.03824961	1
PhysarumPolycephalum	95.2 ± 8.47	0.08993015	0
BarkBeetle	100 ± 0	-	-
	50 nodes, 30 queries, 2% replication		
Gnutella	100 ± 0	0.16846945	0
k-Walker	88.7 ± 6.63	2.3916E-04	1
AntNet	15.9 ± 29.22	4.7646E-04	1
SMP2P	73.3 ± 11.45	1.8240E-06	1
PhysarumPolycephalum	10.7 ± 24.35	1.1501E-09	1
BarkBeetle	98.9 ± 2.42	-	-
	50 nodes, 30 queries, 16% replication		
Gnutella	100 ± 0	-	0
k-Walker	100 ± 0	-	0
AntNet	95.6 ± 9.45	0.15835324	0
SMP2P	97.2 ± 6.27	0.17482693	0
PhysarumPolycephalum	94.3 ± 12.04	0.15161740	0
BarkBeetle	100 ± 0	-	-

Table 6.36: Bark Beetle ANOVA results for Akka (part 1)

	mean ± stdev	p-value	h
	50 nodes, 45 queries, 2% replication		
Gnutella	100 ± 0	-	0
k-Walker	86.6 ± 4.55	2.6476E-08	1
AntNet	24 ± 30.20	2.6357E-07	1
SMP2P	77.4 ± 7.79	3.3184E-08	1
PhysarumPolycephalum	2 ± 0	-	0
BarkBeetle	100 ± 0	-	-
	50 nodes, 45 queries, 16% replication		
Gnutella	100 ± 0	-	0
k-Walker	100 ± 0	-	0
AntNet	97.8 ± 4.49	0.13884044	0
SMP2P	90.9 ± 19.30	0.15330090	0
PhysarumPolycephalum	97.6 ± 4.22	0.08901177	0
BarkBeetle	100 ± 0	-	-
	100 nodes, 10 queries, 1% replication		
Gnutella	100 ± 0	-	0
k-Walker	83 ± 9.49	2.2463E-05	1
AntNet	26 ± 23.66	1.0607E-08	1
SMP2P	76 ± 21.71	0.00257593	1
PhysarumPolycephalum	14 ± 15.78	1.2314E-12	1
BarkBeetle	100 ± 0	-	-
	100 nodes, 10 queries, 16% replication		
Gnutella	100 ± 0	-	0
k-Walker	100 ± 0	-	0
AntNet	93 ± 9.49	0.03142916	1
SMP2P	99 ± 3.16	0.33056493	0
PhysarumPolycephalum	90 ± 9.43	0.00353274	1
BarkBeetle	100 ± 0	-	-
	100 nodes, 30 queries, 1% replication		
Gnutella	100 ± 0	0.04176914	1
k-Walker	82.6 ± 5.08	2.4205E-05	1
AntNet	20.5 ± 23.10	8.1789E-09	1
SMP2P	74.1 ± 20.08	0.00367583	1
PhysarumPolycephalum	11 ± 15.34	2.6719E-12	1
BarkBeetle	96.1 ± 5.63	-	-
	100 nodes, 30 queries, 16% replication		
Gnutella	100 ± 0	-	0
k-Walker	100 ± 0	-	0
AntNet	93.2 ± 10.59	0.05737187	0
SMP2P	87.4 ± 19.02	0.05061243	0
PhysarumPolycephalum	91.1 ± 10.99	0.01964082	1
BarkBeetle	100 ±0	-	-

Table 6.37: Bark Beetle ANOVA results for Akka. (part 2)

	mean ± stdev	p-value	h
	100 nodes, 60 queries, 1% replication		
Gnutella	100 ± 0	0.00112711	-1
k-Walker	85.5 ± 3.66	2.7077E-07	1
AntNet	22 ± 29.45	2.4453E-07	1
SMP2P	71.3 ± 10.34	5.4235E-07	1
PhysarumPolycephalum	26.6 ± 28.85	2.0888E-05	1
BarkBeetle	96.8 ± 2.62	-	-
	100 nodes, 60 queries, 16% replication		
Gnutella	100 ± 0	-	0
k-Walker	100 ± 0	-	0
AntNet	85 ± 19.16	0.02346355	1
SMP2P	92.1 ± 10.27	0.02562809	1
PhysarumPolycephalum	88.4 ± 14.59	0.02167527	1
BarkBeetle	100 ± 0	-	-
	100 nodes, 90 queries, 1% replication		
Gnutella	100 ± 0	5.0394E-05	-1
k-Walker	83.9 ± 4.63	3.5420E-08	1
AntNet	35.5 ± 37.89	6.0803E-05	1
SMP2P	57.5 ± 11.71	2.6467E-09	1
PhysarumPolycephalum	23.1 ± 24.50	1.5928E-08	1
BarkBeetle	97.8 ±1.32	-	-
	100 nodes, 90 queries, 16% replication		
Gnutella	100 ± 0	-	0
k-Walker	100 ± 0	-	0
AntNet	87.7 ± 12.10	0.00481155	1
SMP2P	91.7 ±12.82	5.5427E-02	1
PhysarumPolycephalum	94.5 ± 11.78	0.15704017	0
BarkBeetle	100 ± 0	-	-
	200 nodes, 20 queries, 0.5% replication		
Gnutella	100 ± 0	5.4177E-07	-1
k-Walker	39.5 ± 8.96	3.2893E-04	1
AntNet	6 ± 15.78	1.1928E-07	1
SMP2P	43.5 ± 26.57	0.04788856	1
PhysarumPolycephalum	7.5 ± 12.08	2.8889E-08	1
BarkBeetle	64 ± 15.06	-	-
	200 nodes, 20 queries, 16% replication		
Gnutella	100 ± 0	-	0
k-Walker	100 ±0	-	0
AntNet	88.5 ± 13.13	0.01265200	1
SMP2P	89.5 ± 21.27	0.13595252	0
PhysarumPolycephalum	80.5 ± 15.89	0.00109567	1
BarkBeetle	100 ± 0	-	-

Table 6.38: Bark Beetle ANOVA results for Akka. (part 3)

	mean ± stdev	p-value	h
	200 nodes, 60 queries, 0.5% replication		
Gnutella	100 ± 0	7.7157E-09	-1
k-Walker	35.3 ± 2.79	4.5842E-12	1
AntNet	3.8 ± 11.00	1.4312E-12	1
SMP2P	47.2 ± 31.84	0.01251720	1
PhysarumPolycephalum	7.3 ± 16.32	4.6180E-10	1
BarkBeetle	75.9 ± 7.55	-	-
	200 nodes, 60 queries, 16% replication		
Gnutella	100 ± 0	-	0
k-Walker	100 ± 0	-	0
AntNet	73.6 ± 15.12	3.0371E-05	1
SMP2P	84.8 ± 26.19	0.08309404	0
PhysarumPolycephalum	89.4 ± 8.64	0.00110203	1
BarkBeetle	100 ± 0	-	-
	200 nodes, 120 queries, 0.5% replication		
Gnutella	100 ± 0	3.3711E-10	-1
k-Walker	40.8 ± 5.12	1.2299E-05	1
AntNet	0.9 ± 2.85	2.8906E-13	1
SMP2P	37.6 ± 20.23	0.00327889	1
PhysarumPolycephalum	4.2 ± 13.28	2.0241E-09	1
BarkBeetle	61.7 ± 9.84	-	-
	200 nodes, 120 queries, 16% replication		
Gnutella	100 ± 0	-	0
k-Walker	100 ± 0	-	0
AntNet	90.6 ± 9.59	0.00620180	1
SMP2P	85.7 ± 23.41	0.06929435	0
PhysarumPolycephalum	88.9 ±11.97	0.00888506	1
BarkBeetle	100 ±0	-	-
	200 nodes, 180 queries, 0.5% replication		
Gnutella	99.8 ± 0.42	8.8089E-12	-1
k-Walker	38.7 ± 2.54	5.2457E-11	1
AntNet	1.1 ± 3.14	8.0207E-17	1
SMP2P	22.4 ± 12.41	4.2792E-09	1
PhysarumPolycephalum	8.3 ± 22.06	1.4064E-07	1
BarkBeetle	68.7 ± 6.40	-	-
	200 nodes, 180 queries, 16% replication		
Gnutella	100 ± 0	-	0
k-Walker	100 ± 0	-	0
AntNet	85.9 ± 8.96	9.7986E-05	1
SMP2P	68.6 ± 33.22	0.00787680	1
PhysarumPolycephalum	93.9 ± 4.51	0.00045163	1
BarkBeetle	100 ± 0	-	-

Table 6.39: Bark Beetle ANOVA results for Akka. (part 4)

6.6 Scalability Analysis

This section focuses on load scalability, since it is essential to determine whether P2P network environments can handle network grows.

To evaluate the scalability of distributed systems, Jogalekar et al. [19] provide the following metric:

$$\psi = \frac{F(\lambda_2, QoS_2, C_2)}{F(\lambda_1, QoS_1, C_1)} \tag{6.1}$$

where F is a function evaluating the performance, λ is the throughput, *QoS* are quality of service parameters, and C is the cost.

[19] introduces a strategy for scaling, which is defined by scaling variables to specify a scaling path and a scaling factor k. $\psi(k)$ behaves differently in different situations as Figure 6.7 illustrates.

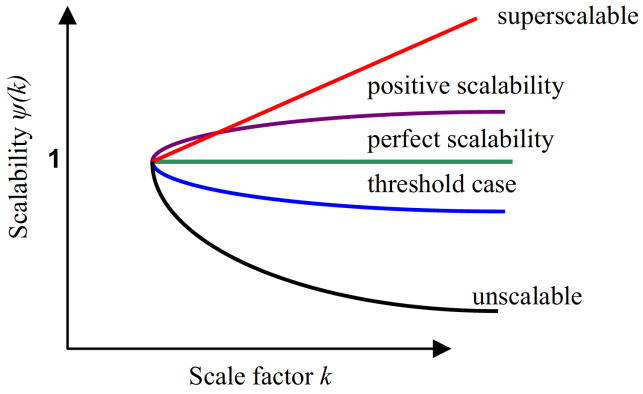


Figure 6.7: Scaling Behavior [19].

In this thesis, Gnutella Flooding, k-Walker, AntNet for P2P, SMP2P, Physarum Polycephalum Slime Mold for P2P and Bark Beetle for P2P have to be analyzed regarding resources, performance and load. To that end, a specialization of Equation 6.1 proposed by Šešum-Čavić et al. [27] is used:

$$P = \frac{1}{M} \tag{6.2}$$

where M represents the average messages for each node, and $M \neq 0$ [27].

The performance *P* becomes increasingly worse, the more messages a single node has to process. Following this, the performance decreases on an increasing demand on a node and increases on a decreasing demand on a node [27].

P(L, R) defines the actual performance, where *L* represents the load and *R* represents the resource. The load is represented by the quantity of queries and the resource is represented by the network size [27]. The following example illustrates this further:

Example 6.2: In Table 6.14 the first row has 50 nodes, 5 queries, and 8 messages per node on average. Thus:

$$P(5, 50) = \frac{1}{8}$$

Equation 6.3 shows the adapted version of Equation 6.1 presented in [27] and defines the load scalability.

$$\psi(k) = \frac{P(kL, kR)}{P(L, R)} \tag{6.3}$$

where k represents the scaling factor and P represents the performance.

To have positive scalability the performance should not decrease, if load and resources are increased by factor *k*. Furthermore, the system does not scale, if the new performance is less than the original one.

The scalability of Gnutella Flooding, k-Walker, AntNet for P2P, SMP2P, Physarum Polycephalum Slime Mold for P2P and Bark Beetle for P2P is evaluated as follows. Since the test cases from Section 6.3 are executed on a network with 50, 100 and 200 nodes, the resources *R* are increased by factor 2 and 4. Additionally, for a network size of 50 the initial load L is 5, 15, 30 and 45 queries, which is increased by factor 2 and 4. Afterwards, the results for all six algorithms are compared. The results for the Peer Model are based on Tables 6.12, 6.13, 6.14, 6.15, 6.16 and 6.17 and are shown in Tables 6.40 and 6.41, where the results for 1 replica are presented in 6.40 and results for 16% replication are presented in 6.41. The results for Akka are based on Tables 6.18, 6.19, 6.20, 6.21, 6.22 and 6.23 and are shown in Tables 6.42 and 6.43, where the results for 1 replica are presented in 6.42 and the results for 16% replication are presented in 6.43.

Initial Load k=1	**Gnutella**		**k-Walker**		**AntNet**		**SMP2P**		**Physarum Polycephalum**		**Bark Beetle**	
	k=2	**k=4**	**k=2**	**k=4**	**k=2**	**k=4**	**k=2**	**k=4**	**k=2**	**k=4**	**k=2**	**k=4**
5	0.50	0.27	1.00	0.90	0.89	0.89	0.56	0.42	1.00	1.00	0.80	1.14
15	0.53	0.28	1.05	0.99	1.04	1.00	0.58	0.48	0.97	0.91	1.00	0.96
30	0.53	0.43	0.99	0.95	0.95	0.91	0.72	1.04	0.91	0.91	0.92	0.89
45	0.53	0.29	1.00	0.98	1.00	0.97	0.71	1.07	0.92	0.92	0.94	0.90

Table 6.40: Peer Model scalability for k=2 and k=4, 1 replica

Initial Load k=1	Gnutella		k-Walker		AntNet		SMP2P		Physarum Polycephalum		Bark Beetle	
	k=2	k=4	k=2	k=4	k=2	k=4	k=2	k=4	k=2	k=4	k=2	k=4
5	0.89	0.57	1.54	1,33	1.33	1.33	0.77	0.63	1.50	1.50	2.43	3.40
15	0.50	0.28	1.27	1.12	1.35	1.18	0.56	0.58	1.21	1.05	1.07	1.07
30	0.56	0.27	1.02	1.01	0.89	0.93	0.58	0.96	1.07	0.98	0.93	0.93
45	0.50	0,26	0.93	0.93	1.00	0.92	0.61	0.83	0.82	0.86	0.89	0.91

Table 6.41: Peer Model scalability for k=2 and k=4, 16% replication

Initial Load k=1	Gnutella		k-Walker		AntNet		SMP2P		Physarum Polycephalum		Bark Beetle	
	k=2	k=4	k=2	k=4	k=2	k=4	k=2	k=4	k=2	k=4	k=2	k=4
5	0.48	0.24	1.04	1.00	1.00	1.00	0.65	0.63	1.11	1.11	0.80	1.14
15	0.50	0.25	1.05	0.99	1.03	1.00	0.77	0.61	1.03	1.00	1.00	0.96
30	0.50	0.20	0.98	0.94	0.94	0.97	0.72	0.87	0.98	0.98	0.98	0.94
45	0.50	0.25	1.00	0.98	1.00	1.00	0.84	1.08	0.95	1.00	0.98	0.96

Table 6.42: Akka scalability for k=2 and k=4, 1 replica

Initial Load k=1	Gnutella		k-Walker		AntNet		SMP2P		Physarum Polycephalum		Bark Beetle	
	k=2	k=4	k=2	k=4	k=2	k=4	k=2	k=4	k=2	k=4	k=2	k=4
5	0.46	0.22	1.62	1.50	1.80	1.50	0.91	0.51	1.17	1.17	2.38	4.75
15	0.47	0.23	1.29	1.14	1.09	1.00	0.95	0.54	1.37	1.13	1.15	1.07
30	0.48	0.23	1.00	0.99	0.96	1.00	0.78	0.64	1.00	1.00	1.00	1.00
45	0.48	0.23	0.93	0.93	0.85	0.85	0.81	0.78	0.84	0.82	0.97	0.97

Table 6.43: Akka scalability for k=2 and k=4, 16% replication

The scalability of Bark Beetle is similar in the Peer Model and in Akka. For 1 replica, the scalability is close to 1 with $\psi(2)$ for initial load 5 having the lowest value of 0.80. But for $\psi(4)$ with the same initial load of 5, the scalability is the highest with 1.14. All other values are between 0.89 and 1 with $\psi(4)$ only having slightly worse values than $\psi(2)$.
In case of 16% replication, the scalability values are even better. The best results are achieved for initial load 5, where $\psi(2)$ is between 2.43 and 2.38 and $\psi(4)$ is between 3.40 and 4.75. For initial load 15, the values are slightly greater than 1. For the remaining loads, the scalability is very close to 1 with values between 0.89 and 1, where the values for scaling factor 2 and 4 are almost identical.

Physarum Polycephalum shows similar results to Bark Beetle in the Peer Model and in Akka. In the case of 1, replica and initial load 5, Physarum Polycephalum has a better scaling for $\psi(2)$.
In the case of 16% replication and initial load 5, Physarum Polycephalum is worse than Bark Beetle for both $\psi(2)$ and $\psi(4)$, although still greater than 1. Additionally, the values for initial

load 45 are slightly worse in Physarum Polycephalum with values between 0.82 and 0.86 as opposed to Bark Beetle with values between 0.89 and 0.97. Thus, both search algorithms proposed in this thesis show very good scalability as the initial load grows.

In comparison to Gnutella, both Bark Beetle and Physarum Polycephalum show better scalability in all cases. Only in the Peer Model with 16% replication for initial load 5 the scalability is good with $\psi(2)$ = 0.89. Excluding the initial load 5 with 16% replication from the Peer Model results, the highest value for $\psi(2)$ is 0.56 and 0.43 for $\psi(4)$. The lowest value for both Bark Beetle and Physarum Polycephalum is 0.80.

In case of 1 replica, the values for k-Walker are between 0.90 and 1.05. In case of 16% replication, the values are between 0.93 and 1.62. Thus, the scalability is similar to both Bark Beetle and Physarum Polycephalum. Only for 1 replica and initial load 5 the scalability for Bark Beetle is significantly better than k-Walker.

In comparison to AntNet, both Bark Beetle and Physarum Polycephalum have very similar scalability. Only in the case of 16% replication and initial load 5, Bark Beetle is much better, although the scalability of AntNet is also very good with values between 1.33 and 1.80.

In general, SMP2P has worse scalability than both Bark Beetle and Physarum Polycephalum, with only some exceptions. In the Peer Model, SMP2P is slightly better in case of 1 replica and initial load of 30 and 45 for $\psi(4)$. The same is true in Akka in case of 1 replica and initial load 45 for $\psi(4)$. Thus, it can be argued that SMP2P has slightly better scalability for 1 replica and high load, although both Bark Beetle and Physarum Polycephalum achieved very good results for $\psi(4)$ with initial load 45 with values between 0.90 and 1.00.

CHAPTER 7

Future Work and Conclusion

This chapter discusses possible future improvements to the frameworks, the adapted search algorithm Physarum Polycephalum Slime Mold for unstructured P2P and the newly created search algorithm Bark Beetle for unstructured P2P. Afterwards, a final conclusion to the master thesis is drawn.

7.1 Future Work

- Since Bark Beetle for unstructured P2P and Physarum Polycephalum for unstructured P2P have only been benchmarked in a static P2P network environment, their adaptiveness to dynamic environments with high peer churn should also be analyzed.
- The benchmarking results for Bark Beetle for unstructured P2P and Physarum Polycephalum for unstructured P2P show, that both algorithms perform well as the network size and load increases. To ensure that this trend continues as the network size and load increases further, additional benchmarks for larger network instances should be carried out.
- To improve the usability of the framework a graphical user interface to configure and execute benchmarks in the framework would be helpful. Additionally, an option to automatically create plots could increase the user experience significantly as it would save a lot of time.

7.2 Conclusion

In this thesis the need to evaluate and compare search algorithms for unstructured P2P networks with each other by using standard metrics is addressed. Therefore, two goals are achieved by this thesis.

The first goal of the thesis is to implement two frameworks for benchmarking of search algorithms for unstructured P2P networks. The first framework is implemented based on the Actor Model [16] using Akka and the second one based on the Peer Model [12] using its Java implementation. Different search algorithms can be plugged into the frameworks.

The benchmarking frameworks show to fulfill all defined requirements. They provide a clearly structured architecture that allows new users to understand the basic concepts very quickly. Additionally, both frameworks offer extensive configurability with provided default values that fit many scenarios. Furthermore, generic interfaces allow easy exchangeability of different algorithms.

The second goal is to use the created frameworks to develop a new algorithm for distributed search in P2P networks based on the collective feeding of bark beetles [22]. Additionally, an adaption of the already existing Physarum Polycephalum algorithm [9] to search in fully distributed P2P networks is developed. Both algorithms are intelligent search algorithms that use software agents to learn about their environment in order to resolve search requests. The resulting algorithms are implemented, evaluated and compared to the non-intelligent algorithms Gnutella Flooding and k-Walker as well as the intelligent algorithms AntNet and SMP2P using the created frameworks.

Both Bark Beetle and Physarum Polycephalum show to have very good scalability regarding growing network size and load. They scale equal to or better than the remaining evaluated algorithms. Two strategies for replication are used to evaluate the six algorithms: for low replication only 1 node has the required resource and for high replication 16% of the nodes in the network have the required resource. In case of 1 replica, Physarum Polycephalum outperforms all other algorithms only matched by AntNet in terms of absolute time. Bark Beetle performs only slightly worse than Physarum Polycephalum. In case of 16% replication, Bark Beetle performs better than most of the algorithms with only k-Walker slightly outperforming it and is closely followed by Physarum Polycephalum. In terms of average messages per node, Bark Beetle shows the best results closely followed by Physarum Polycephalum in all cases. In terms of success rate, Bark Beetle shows very good results performing only slightly worse than Gnutella in case of 1 replica and being on par with it in case of 16% replication. On the other hand, Physarum Polycephalum has the worst success rate together with AntNet in case of 1 replica. In case of 16% replication, it performs significantly better alongside AntNet, matching SMP2P at times, but is outperformed by the remaining algorithms non the less.

This thesis shows that it is possible to provide frameworks for benchmarking of search algo-

rithms for unstructured P2P networks based on the Actor Model and Peer Model. Additionally, two new search algorithms are provided that show to perform very well.

Bibliography

[1] Akka. Documentation. `https://akka.io`, 2017. Accessed: 2017-10-22.

[2] Google Compute Engine. `https://cloud.google.com/compute/`, 2017. Accessed: 2017-07-12.

[3] Google Compute Engine Machine Types. `https://cloud.google.com/compute/docs/machine-types`., 2017. Accessed: 2017-07-12.

[4] A. L. Barabási, R. Albert. Emergence of scaling in random networks. In *Science*, volume 286, pages 509–512. American Association for the Advancement of Science, 1999.

[5] A. Oram. *Peer-to-Peer : Harnessing the Power of Disruptive Technologies*. O'Reilly Media, 1 edition, 2001.

[6] Mauro Birattari. *Tuning Metaheuristics: A Machine Learning Perspective*. Springer Publishing Company, Incorporated, 1st ed. 2005. 2nd printing edition, 2009.

[7] D. D. Kanev. Decentralized unstructured flat p2p network with streaming content delivery method and user collaboration. Master's thesis, Vienna University of Technology, 2015.

[8] D. R. Monismith Jr. The uses of the slime mold lifecycle as a model for numerical optimization. Stillwater, OK, USA, 2010. Oklahoma State University. AAI3422282.

[9] D. S. Hickey, L. A. Noriega. Insights into information processing by the single cell slime mold physarum polycephalum. In *UKACC Control Conference*, Manchester, UK, 2008.

[10] D. Stutzbach, R. Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, IMC '06, pages 189–202, New York, NY, USA, 2006. ACM.

[11] E. K. Lua, H. Yu, J. Buford. *P2P Networking and Applications*. M. Kaufmann, 1 edition, 2008.

[12] E. Kühn, S. Craß, G. Joskowicz, A. Marek, T. Scheller. Peer-based programming model for coordination patterns. In *Coordination Models and Languages: 15th International Conference, COORDINATION 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings*, pages 121–135, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[13] F. Sharifkhani, M. R. Pakravan. A new metric for comparison of p2p search algorithms. In *2012 Seventh International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, pages 191–195. IEEE Computer Society, Nov 2012.

[14] G. A. 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, November 2004.

[15] J. F. Buford, H. Yu. Peer-to-peer networking and applications: Synopsis and research directions. In *Handbook of Peer-to-Peer Networking*, pages 3–45, Boston, MA, 2010. Springer US.

[16] J. Goodwin. *Learning Akka*. Packt Publishing, 1 edition, 2015.

[17] M. Casadei, R. Menezes, M. Viroli, R. Tolksdorf. A self-organizing approach to tuple distribution in large-scale tuple-space systems. In R. H. Katz D. Hutchison, editor, *Self-Organizing Systems*, volume 4725 of *Lecture Notes in Computer Science*, pages 146–160. Springer Berlin Heidelberg, 2007.

[18] M. López-Ibáñez, J. Dubois-Lacoste, T. Stützle, M. Birattari. The irace package: Iterated racing for automatic algorithm configuration. In *Operations Research Perspectives*, volume 3, pages 43 – 58. Elsevier, 01 2011.

[19] P. Jogalekar, M. Woodside. Evaluating the scalability of distributed systems. In *IEEE Trans. Parallel Distrib. Syst.*, volume 11, pages 589–603, Piscataway, NJ, USA, June 2000. IEEE Press.

[20] Q. Lv, P. Cao, E. Cohen, K. Li, S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, pages 84–95, New York, NY, USA, 2002. ACM.

[21] S. Androutsellis-Theotokis, D. Spinellis. A survey of peer-to-peer content distribution technologies. In *ACM Comput. Surv.*, volume 36, pages 335–371, New York, NY, USA, December 2004. ACM.

[22] D. Sauvard. *General Biology of Bark Beetles*, pages 63–88. Springer Netherlands, Dordrecht, 2004.

[23] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. In *Soviet Physics Doklady*, volume 10, page 707. Elsevier, 1966.

[24] V. Vernon. *Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka*. Addison-Wesley Professional, 1st edition, 2015.

[25] V. Šešum-Čavić, E. Kühn. A swarm intelligence appliance to the construction of an intelligent peer-to-peer overlay network. In *2010 International Conference on Complex, Intelligent and Software Intensive Systems*, pages 1028–1035. IEEE Computer Society, Feb 2010.

[26] V. Šešum-Čavić, E. Kühn. Algorithms and framework for comparison of bee-intelligence based peer-to-peer lookup. In *Advances in Swarm Intelligence: 4th International Conference, ICSI 2013, Harbin, China, June 12-15, 2013, Proceedings, Part I*, pages 404–413, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[27] V. Šešum-Čavić, E. Kühn, D. Kanev. Bio-inspired search algorithms for unstructured p2p overlay networks. volume 29, pages 73 – 93. Elsevier, 2016.