Contract nº 507953

WP 24: DBE Implementation
D24.8: The DBE P2P SOA

Project funded by the European Community under the "Information Society Technology" Program

Contract Number: 507953
Project Acronym: DBE
Title: Digital Business Ecosystem

Deliverable Nº: 24.8
Due Dates: 08/2006
Delivery Date: 09/2006

Short Description: This document is a successor of the deliverable D24.3 "DBE Peer-to-Peer Architecture Design". It covers a detailed design and implementation of the P2P framework and its integration into the overall servent architecture. Further, the topologies are evaluated and issues discussed in detail.

Author: Trinity College Dublin (TCD)
Partners Contributed:
Made Available To: Public

Versioning

Version	Date	Author, Organisation	Description
0.1	April - June 2006	Jan Sacha, René Meier	Gradient Topology
0.2	April - June 2006	Bartosz Biskupski, René Meier	DHT
0.3	07.07.2006	Dominik Dahlem	Document Structure
0.4	July 2006	Dominik Dahlem, Jan Sacha, Bartosz Biskupski	Design, Implementation
0.5	August 2006	Jan Sacha	GT, Bootstrap
0.6	August 2006	Bartosz Biskupski	DHT, Bootstrap
0.7	29.08.2006	Dominik Dahlem, Jan Sacha, Bartosz Biskupski	Review-Cycle
0.8	September 2006	Jan Sacha, Bartosz Biskupski, Dominik Dahlem	Evaluation, Conclusion
0.9	11.09.2006	Jan Sacha, Bartosz Biskupski, Dominik Dahlem, René Meier	Review-Cycle
1.0	19.09.2006	Dominik Dahlem, Jan Sacha, René Meier	Version submitted to the reviewers
1.1	16.10.2006	Jan Sacha, Bartosz Biskupski, Dominik Dahlem, René Meier	Final version including reviewer's comments
1.2	31.10.2006	Jan Sacha, Bartosz Biskupski, Dominik Dahlem, René Meier	Revised final version
1.3	14th of November	Dominik Dahlem	Revised the style of the header/footer

Quality Check:
1st Internal Reviewer: Miguel Vidal, SUN
2nd Internal Reviewer: David Singer, IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120, USA. E-mail: singer@almaden.ibm.com

Contents

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

This chapter presents an introduction to the document. It motivates (see Section 1.1) the topic and outlines its scope (see Section 1.2) with regards to its predecessor deliverable D24.3 “DBE Peer-to-Peer Architecture Design”.

1.1 Motivation

The previous deliverable D24.3 “DBE Peer-to-Peer Architecture Design” covered initial works related to task C57 “P2P Architecture and Service-Oriented Routing” and task C18 “Distributed Identity Service”. The milestones that were associated with it include M24.18 “P2P Architecture Document”, M24.19 “Design of a Service-Oriented Routing Protocol”, and M24.7 “First release of the Distributed Identity Framework and Services”.

The previous document analysed open platform characteristics for an unlimited number of SMEs and set forth requirements that allow an open platform to flourish using decentralised P2P overlays instead of centralised approaches. The P2P overlays provide a self-organising mechanism for service placement, in particular for the Knowledge Base and Semantic Registry (KB/SR). By using an unstructured P2P topology for service placement based on a node’s utility a heuristic search had to be employed to allow the discovery of those services from any node via a multi-hop protocol. Moreover, the unstructured gradient topology is accompanied by a structured Distributed Hashtable topology for Service Proxy placement. The special feature of a DHT is the guarantee to find a Service Proxy, if it is deployed in the system. Both topologies in combination provide a unique way of delivering a self-organising P2P service-oriented architecture, that is the DBE.

The last version of the document was pushing the boundary of what is technically feasible, especially since the research in this area is in its early stages. This document will give a revised design and implementation of the topologies. Further, detailed discussions are provided to expand our architecture into the next generation DBE that may include more topologies to satisfy specific tasks.

1.2 Scope

Since the last publication of D24.3, our research has advanced resulting in more stable and reliable algorithms to satisfy a business environment that self-organises into a

network structure purely by a node's status without any directed (possibly malicious) placement. This document will reflect our advanced research and thereby addresses the comments made during the Tampere review.

In particular, the reviewers raised the following issues:

- General
  - Remove the need of the central DBE portal for bootstrap purposes;
  - More advanced simulation to justify the need and benefit for continuous topology optimisation by rearranging existing connections;
- Gradient Topology
  - Differentiate the usefulness of a node in the network through multiple utility functions;
  - Clarify the concept and the role of super-peers to address the politics of the network;
- DHT Topology
  - Justify the use of the DHT with simulation results;

This document will address these issues and provides solutions to satisfy the ambitious goal of the project to provide a conceptually complete P2P SOA.

Chapter 2

Design of the Service-Oriented P2P Architecture

In this chapter we present the design of the service-oriented P2P architecture. First the abstract design of the Servent with the P2P architecture as a core service is presented followed by use cases with respect to the SOA activities deploy, search, and consume and also taking into account the perspectives of the consumer and the service provider. Section 2.3 and Section 2.4 expand on the designs of the Gradient Topology and the Distributed Hashtable respectively. Section 2.5 outlines solutions to the bootstrap problem common to self-organising P2P systems. Finally, Section 2.6 outlines how possible extensions fit into the overall frame of the P2P architecture presented in this document.

2.1 Architecture

Figure 2.1 gives a high-level overview of the core components of the Servent architecture with an emphasis on P2P integration as a core service. The servent maintains loosely coupled relationships to its core services, such as KB/SR, workflow management, P2P overlays, and identity among others.

The P2P core service component outlines some high-level interfaces that are involved in providing a service-oriented P2P service registry facility within the DBE middleware platform. It exposes an abstract API that hides the implementation and topology detail in order to deploy, discover, and undeploy DBE services. Section 2.3 and Section 2.4 will provide more insights into the respective designs and other APIs that complement the P2P architecture.

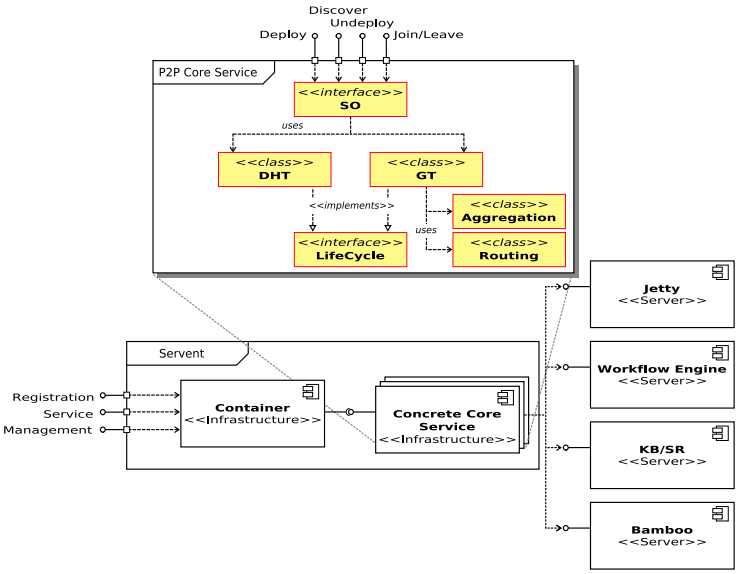

Figure 2.1: Overview of the Servent Architecture

Table 2.1 shows the relationship between the DBE SOA use cases and the DBE P2P architecture components.

Component	Use Cases
Gradient Topology Overlay	KB/SR Instance Discovery
DBE Knowledge Base and Semantic Registry (KB/SR)	Service Registration, Deregistration, and Discovery
Distributed Hashtable Overlay (DHT)	Service Proxy Upload, Download, and Renew
DBE Execution Environment (ExE)	Service Container

Table 2.1: Relation between the DBE use cases and the DBE P2P architecture components

The DBE Knowledge Base and Semantic Registry (KB/SR), and the DBE Execution Environment (ExE), are both maintained by other DBE partners and do not belong to the DBE P2P architecture. For this reason, their design and implementation will not be described in this document and only selected features will be mentioned. The focus of this document is the two P2P overlays, gradient topology and distributed hash table.

2.2 Usage Scenarios

The following sections give an overview of the functional model of the service-oriented P2P architecture. In general it follows the basic principles of service-oriented architec-

tures deployment, discovery, and consumption (see Figure 2.2 and [1, 10]). In particular the functional model takes into account the interactions between those principles and our P2P architecture. A service in the context of SOAs is a self-contained black box component that can itself be assembled by other services into more complex business scenarios. As such, we do not distinguish between complex or simple services at the functional level, but rather regard those as a deployable unit.

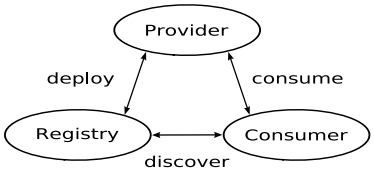

Figure 2.2: Basic SOA Principles

In order to provide a complete life-cycle of a service, the undeployment activity is considered as well. Further, the use cases involved in cleaning up an open service-oriented middleware, such as the DBE, are important aspects of the manageability of the P2P topologies.

2.2.1 Deploying Services

The functionality of deploying services in the DBE environment is reserved to service providers. A service provider has access to the DBE middleware platform, called Servent, in order to make his services or the services he is responsible for available to the public. Figure 2.3 presents the use cases involved in deploying a service. The internals of the deployment process, i.e., the interaction with the core services, in particular the P2P topologies and the KB/SR service are shielded from the service provider.

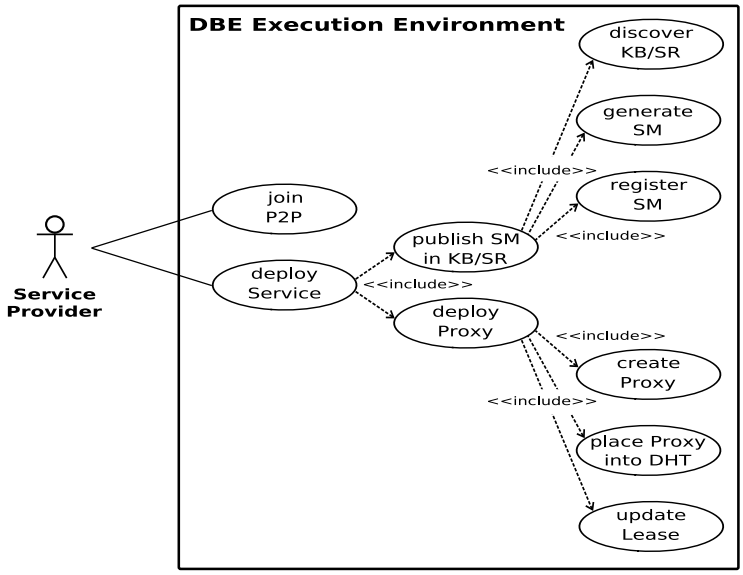

Figure 2.3: Deploy Use Case

Use Case	**join P2P**
Actor:	Service Provider
Pre-Conditions:	A servent installation is required in order to participate in the service-oriented P2P network.
Post-Conditions:	
Description:	By joining the P2P a servent participates in the storage capability of the DHT topology. Once a servent joins, it can use the DHT topology to put and retrieve data from it. Analogue for the Gradient Topology which accompanies the DHT topology. Thus, joining the P2P implies a dual join procedure to both, the Gradient Topology and the DHT.
Dependencies:	
Partner(s) Responsible:	TCD

Use Case	deploy Service
Actor:	Service Provider
Pre-Conditions:	A DBE Service must have been implemented and packaged using the DBE Studio tools.
Post-Conditions:	
Description:	The deployment of a DBE Service is a process to publish the semantic information about a service and instantiating the implementation of it. Once, the deployment process is completed successfully, the service is available in the DBE. The deployment entails the installation, configuration, and execution of the service (ICE principle).
Variations:	In general the result of a successful deployment process is a service that can be invoked by DBE consumers. In the specific context of the DBE the deployment process is split into two phases: publishing and deployment. Publishing entails the registration of the service models with the KB/SR, whereas the deployment installs the service implementation into the servent and uploads a service proxy with the DHT topology. A special case arises when the service provider does not want to deploy a complete service, but instead prefers to publish his models (BML and SDL) first. The DBE generally allows a loose coupling between models that represent a service and a service instance that can be invoked.
Dependencies:	register SM in KB/SR, deploy Proxy
Partner(s) Responsible:	TCD, SUN

Use Case	publish SM in KB/SR
Actor:	Service Provider
Pre-Conditions:	The DBE Service has to pass the syntactical and semantical validation of the package.
Post-Conditions:	Upon completing the registration process, the DBE service is only made public, but not yet available for consumption. In order to complete the deployment process, the service has to be installed and registered with the P2P topology.
Description:	Relevant data for the registration process are the BML, and the SDL models of the DBE service. Both models can be queried once the registration is successfull.
Dependencies:	discover KB/SR, generate SM
Partner(s) Responsible:	TUC

Use Case	discover KB/SR
Actor:	Service Provider
Pre-Conditions:	
Post-Conditions:	
Description:	Unlike many SOAs, the DBE P2P architecture is completely decentralised. As such an address of a KB/SR replica has to be obtained using the Gradient Topology and Gradient Search (described later in the document).
Dependencies:	
Partner(s) Responsible:	TCD

Use Case	generate SM
Actor:	Service Provider
Pre-Conditions:	Valid BML and SDL models
Post-Conditions:	
Description:	Both models, BML and SDL, are extracted from the DBE service package and assembled in a Service Manifest (SM).
Dependencies:	
Partner(s) Responsible:	SUN, Soluta

Use Case	register SM
Actor:	Service Provider
Pre-Conditions:	A Service Manifest must be generated
Post-Conditions:	
Description:	The KB/SR services expose an API that allows the registration of compliant models, such as the Service Manifest. The registration process involves storing the SM into the database provided by the KB/SR.
Dependencies:	
Partner(s) Responsible:	TUC

Use Case	deploy Proxy
Actor:	Service Provider
Pre-Conditions:	Requires successful registration with the KB/SR
Post-Conditions:	Trigger a lease update so that the proxy does not expire unless there is a failure in the infrastructure
Description:	Deploying a service into the servent and the P2P infrastructure allows it to be consumed by DBE consumers.
Dependencies:	create Proxy, place Proxy into DHT, update Lease
Partner(s) Responsible:	SUN, TCD

Use Case	create Proxy
Actor:	Service Provider
Pre-Conditions:	
Post-Conditions:	
Description:	For each service the specific service configuration is wrapped into a service proxy. This service proxy “knows” the binding information necessary to consume a service and shields this information transparently from the consumer.
Dependencies:	
Partner(s) Responsible:	SUN

Use Case	place Proxy into DHT
Actor:	Service Provider
Pre-Conditions:	Availability of a service proxy object; Requires the servent to be joined with the DHT
Post-Conditions:	
Description:	The service proxy is put into the distributed hashtable using the DHT topology. The service object is identified by the unique Service Manifest identifier.
Dependencies:	
Partner(s) Responsible:	TCD

Use Case	update Lease
Actor:	Service Provider
Pre-Conditions:	Successful service proxy upload
Post-Conditions:	reset the timer for the lease update
Description:	The DHT topology has a self-cleansing feature that allows dead service proxies to be removed from the DHT ring. The removal occurs when a lease of a service proxy is not updated anymore. That is usually the case when a servent goes unexpectedly or expectedly down.
Dependencies:	
Partner(s) Responsible:	TCD

2.2.2 Discovering Services

Once DBE services are deployed they can be discovered in order to assemble candidate services into a service composition or consumed by a consumer. The discovery mechanism is hidden in order to provide a non-obtrusive user experience. Internally, several core services are utilised to find a service based on some search criteria using the business ontologies, business models, or service definitions. Figure 2.4 shows the use cases involved in discovering a service. The two use cases “join DHT” and “discover KB/SR” are not explained any further, because the previous Section 2.2.1 covers them already.

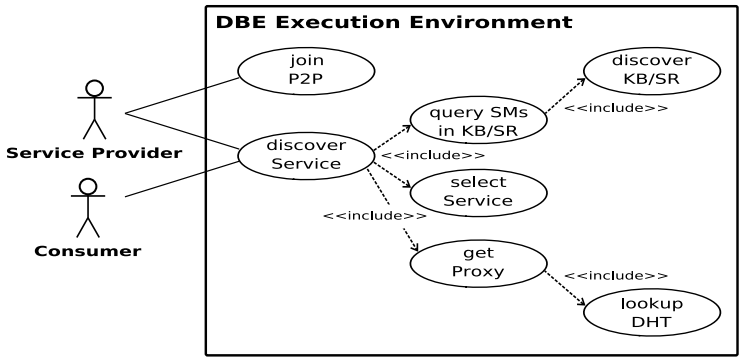

Figure 2.4: Discover Use Case

Use Case	discover Service
Actor:	Consumer, Service Provider
Pre-Conditions:	A service (BML, SDL, and serice instance) must be deployed and available.
Post-Conditions:	
Description:	The process of discovering a service uses both P2P topologies. It returns a service that matches the query criteria. In the case of the Service Provider, the servent must join the P2P network topologies.
Variations:	The actor may not use the discovery mechanism to find a specific service in order to invoke it, but rather he may be interested in searching for a number of services that fulfil his criteria. For example the service provider might be interested in a number of candidate service for a service composition. In that case the service provider only needs the SMs of the respective services.
Dependencies:	query SMs in KB/SR, select Service, get Proxy
Partner(s) Responsible:	SUN, TCD, TUC

Use Case	query SMs in KB/SR
Actor:	Consumer, Service Provider
Pre-Conditions:	
Post-Conditions:	
Description:	A service provider or a consumer identifies some search criteria that is relevant for the search query. The queries are based on semantic information contained in the BML and SDL models (SM). A number of SMs are returned to the user upon which he needs to select which one to consume.
Dependencies:	discover KB/SR, return specific SMs
Partner(s) Responsible:	TUC

Use Case	select Service
Actor:	Consumer, Service Provider
Pre-Conditions:	At least one SM must have been returned by the KB/SR
Post-Conditions:	
Description:	In order to invoke on a specific service a specific service instance has to be selected either by the user or the execution environment, if the user provided specific query information.
Dependencies:	
Partner(s) Responsible:	N/A

Use Case	get Proxy
Actor:	Consumer, Service Provider
Pre-Conditions:	A service must have been chosen.
Post-Conditions:	
Description:	A service proxy is stored in the DHT topology. This topology provides a guaranteed way of retrieving stored proxies, if they are available. A proxy that matches the given identifier is returned.
Dependencies:	lookup DHT
Partner(s) Responsible:	TCD

Use Case	lookup DHT
Actor:	Consumer, Service Provider
Pre-Conditions:	The P2P must be joined
Post-Conditions:	
Description:	The lookup procedure leverages the lookup interface of the P2P architecture in order to find a specific proxy object in the DHT. If there is a service proxy available in the DHT that matches the identifier the search is guaranteed to succeed.
Dependencies:	
Partner(s) Responsible:	TCD

2.2.3 Consuming Services

Upon successful discovery of one or multiple services, they can be consumed by a DBE consumer. The service proxy that is available to a consumer hides the transport-specific and GUI-specific internals and pulls classes from the codebase of the service once they are required. It presents the consumer in most cases with a forms-based GUI

to simplify the interaction with the remote service. Figure 2.5 shows the use cases for service consumption. The previously discussed use case “discover Service” is not covered.

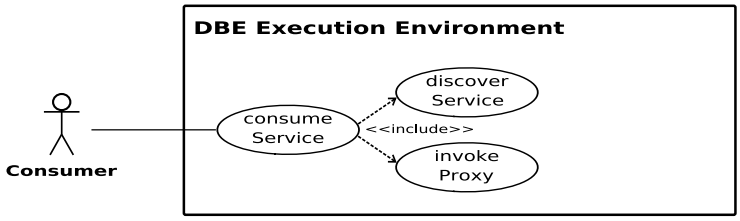

Figure 2.5: Consume Use Case

Use Case	**consume Service**
Actor:	Consumer
Pre-Conditions:	A service must be deployed and accessible.
Post-Conditions:	
Description:	The DBE is an open service-oriented architecture that promotes loose-coupling between and among services and consumers. Any service that is available can be consumed by a DBE consumer. The consumer interacts with the DBE middleware platform that makes a service endpoint available.
Dependencies:	discover Service, invoke Proxy
Partner(s) Responsible:	N/A

Use Case	**invoke Proxy**
Actor:	Consumer
Pre-Conditions:	A discovery procedure must return a valid service proxy.
Post-Conditions:	
Description:	A service proxy is a wrapper that encapsulates the service configuration, such as endpoint and protocol, and it may also be configured to use a number of GUIs. A consumer is unaware of any transport-specific features of the service invocation. The proxy provides the functionality to interact with the remote service.
Dependencies:	
Partner(s) Responsible:	N/A

2.2.4 Undeploying Services

Undeploying services closes the life-cycle ranging from deployment, discovery, to consumption. In the DBE models of services are not deleted in order to allow for evolutionary effects by promoting re-use. Instead, it is encouraged to add new versions of models. Hence, the undeployment (see Figure 2.6) involves removing the service instance from the Servent and the service proxy object from the DHT topology, but not the models that were associated with this service.

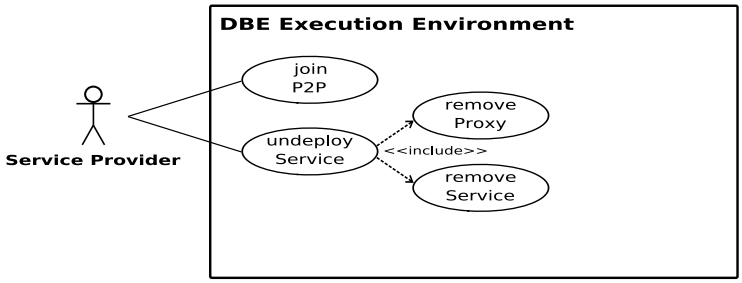

Figure 2.6: Undeploy Use Case

Use Case	**undeploy Service**
Actor:	Service Provider
Pre-Conditions:	A service must have been fully deployed and accessible. If the servent node is not set up behind a NAT gateway, then the servent has to join the P2P network topologies.
Post-Conditions:	The service must be removed in a specific order. First the proxy object is removed from the DHT in order to avoid stale objects, then the service instance is removed from the servent.
Description:	Analogue to the deployment, undeploying a service involves removing the respective parts of a service except the models that are stored in the KB/SR.
Dependencies:	remove Proxy, remove Service
Partner(s) Responsible:	SUN, TCD

Use Case	remove Proxy
Actor:	Service Provider
Pre-Conditions:	
Post-Conditions:	
Description:	The DHT topology provides the functionality to remove a proxy object from the DHT. In doing so, the service is effectively disabled, because consumers cannot download a service proxy object anymore.
Dependencies:	
Partner(s) Responsible:	TCD

Use Case	remove Service
Actor:	Service Provider
Pre-Conditions:	
Post-Conditions:	
Description:	Once the service proxy object is deleted from the DHT topology, the service provider undeploys the service instance from the Servent.
Dependencies:	
Partner(s) Responsible:	TCD

2.3 The Gradient Topology

In this section, we describe the concept of the gradient P2P topology and we outline its main properties. The gradient topology is a P2P topology where each peer is assigned a *utility* value and the highest utility peers are connected with each other and form the so called *core* of the system, while lower utility peers are located gradually farther from the core. Peer utility is an application specific metric that measures the ability of a peer to provide and maintain system services. The highest utility peers, located in the core, are hence the most suitable for maintaining system services or system data. Figure 2.7 shows a picture of a sample gradient topology where the peer utility has been marked by the colour intensity.

The key property of the gradient topology is that it enables a very efficient search algorithm, called *gradient search*, for high utility peer discovery. Given that the utility metric can be arbitrarily defined by the system designer, the gradient topology allows efficient discovery of peers with any desired characteristics.

We have designed, implemented, and tested gradient topology software that allows peers to create and maintain the gradient topology structure, to calculate their own

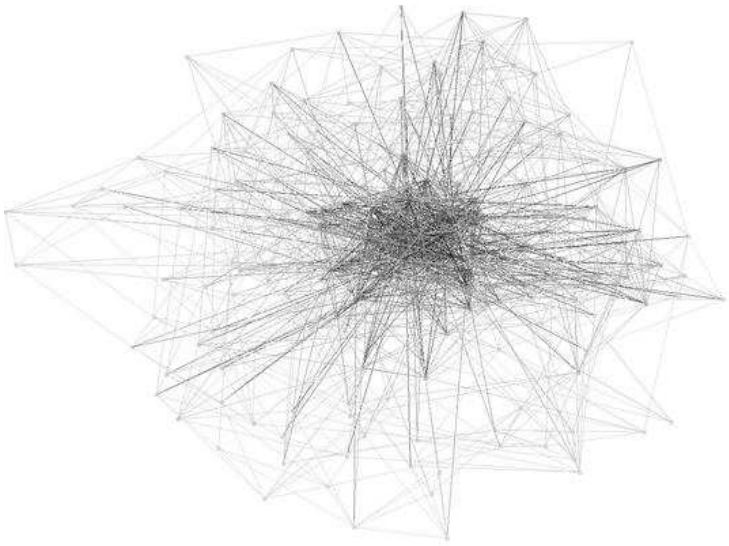

Figure 2.7: Visualisation of a sample gradient topology.

utility, to estimate system-wide utility properties, such as the distribution of peer utility in the system, and to effectively search for high utility peers in the system.

The main motivation for developing the gradient topology was the fact that the gradient topology enables an efficient implementation of a service registry, such as the DBE Knowledge Base (KB) and Semantic Registry (SR). It is assumed that the service registry is distributed between a number of peers in the system for reliability and performance reasons. Hence, there are two types of peers: *core-peers* (also called super-peers) that host registry replicas, and *ordinary peers* that maintain no replicas. By selecting only the best performing, most reliable peers for the registry maintenance, the system can improve the overall registry stability a performance. Low utility peers, if used for service registry replication and maintenance, could degrade the performance of the entire system. Hence, gradient topology is used to enable efficient selection of the most suitable peers for the service registry maintenance.

The suitability of a peer to become a super-peer is measured as the peer's *utility*. A utility *threshold* is defined as a criteria for super-peer selection, i.e., peers with their utility above the specified threshold become super-peers. Figure 2.8 shows a sample P2P gradient topology with service registry replicas stored by the peers in the core of the topology. The core, i.e., peers maintaining the registry, are determined by the replica placement threshold. Gradient search is used by ordinary peers to discover peer above the threshold that host the registry replicas.

Given that the utility of each peer in the topology can be calculated by a common function, the selection of super-peers becomes a question of how to set the super-peer utility threshold. In one possible approach, the threshold may be defined as an absolute utility value. However, in many other applications, before a peer attempts to determine the super-peer utility threshold, the peer needs to estimate the distribution of peer utility in the system in order to know what constitutes high utility in a running system. For example, if an application requires the selection of the most stable peers in the system,

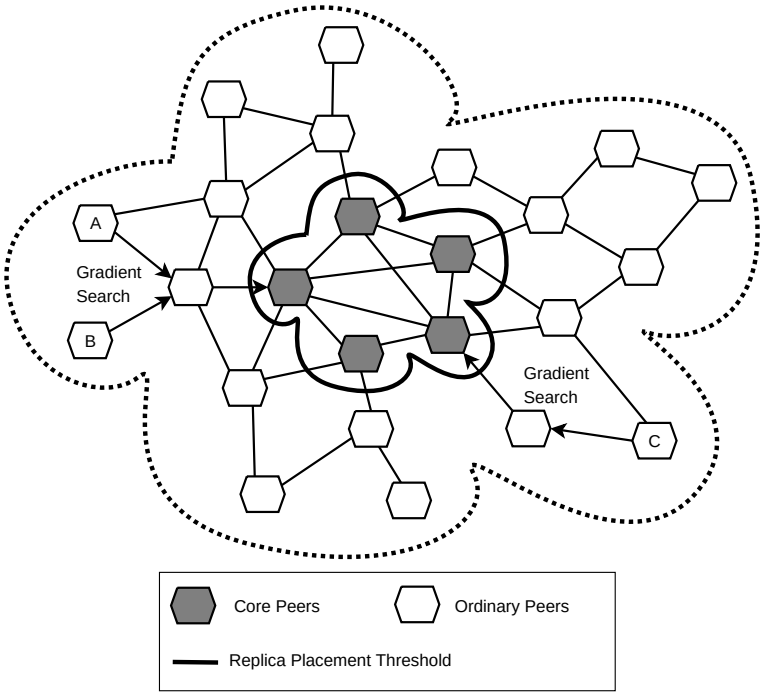

Figure 2.8: Service registry replication and discovery in the gradient topology.

it needs to learn the peer stability characteristics before it can decide on the stability threshold for a super-peer. Otherwise, if the super-peer threshold is static (hardwired), it may happen that no peer in the system satisfies the criteria, or that the threshold is very low and hence sub-optimal. Moreover, due to the system's dynamism, the super-peer selection criteria has to be continuously adapted to the system's current state and peer availability.

Peers in the gradient topology periodically perform a decentralised *aggregation* algorithm that allows the calculation of global system properties, such as the number of peers in the system and the minimum, maximum peer utility in the system, and peer *utility histogram* that approximates peer *utility distribution* in the system. The system-wide aggregates are used by peers to calculate the super-peer utility threshold. The aggregates are calculated periodically at runtime, which allows peers to adaptively adjust the super-peer threshold during system operation.

The algorithms offered by the gradient topology can be used by the DBE KB/SR for performing two tasks:

- Gradient topology allows the KB/SR implementation to select the most suitable peers for hosting the KB/SR replicas.
- Gradient topology allows ordinary peers (users) to discover instances (i.e., replicas) of the DBE KB/SR.

The KB/SR replica membership management and the KB/SR replica synchronisation are handled by higher level protocols that are part of the KB/SR implementation and hence are not part of the P2P architecture. We expect that the number of KB/SR replicas

is considerably lower that the total number of peers in the system, and hence, it is possible for each KB/SR replicas to maintain a list of all KB/SR replicas in the system. This is further facilitated by the fact that the replicas are hosted by high utility, stable, and well-connected peers. The replica list may be reactively updated and propagated between peers whenever a replica is created or removed, or a gossip-based approach may be adopted, where high utility peers located at the core of the gradient topology periodically exchange and update their replica lists.

Figure 2.9 shows the general procedure of KB/SR instance discovery by peers. A peer determines whether it stores a local KB/SR replica, and returns a local replica if one exists. If no local replica is maintained, the peer calculates the KB/SR replica threshold using the utility histogram generated by the aggregation algorithm and performs gradient search in order to discover a peer above the specified utility threshold that stores a KB/SR replica.

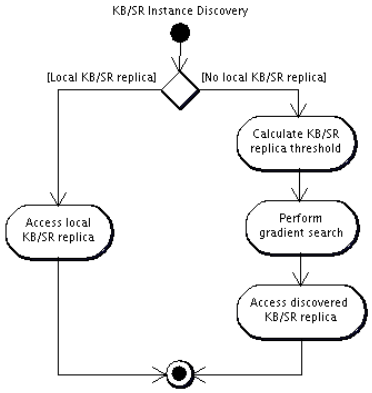

Figure 2.9: KB/SR instance discovery algorithm.

Figure 2.10 shows a sequence diagram of interactions between components of the gradient topology when a peer attempts to discover a KB/SR replica.

In order to determine whether a peer should become a super-peer (i.e., a peer maintaining a KB/SR replica), each peer periodically calculates it's own utility, using the *utility function*, and compares it with the threshold for KB/SR replica placement, calculated using the *threshold function*. It is important to note here, that both utility and threshold functions must be calculated in the same way by all peers in the system. This is required to preserve system integrity.

This poses some risks to potential attacks by a hostile peer which could degrade system functionality by claiming a very high utility. The hostile peer may respond to heuristic search requests, but never completes actual work. Possible solutions to this problem include:

- **Strong Identities**: A peer that would like to participate in the registry placement mechanism is required to present a strong identity to the network.

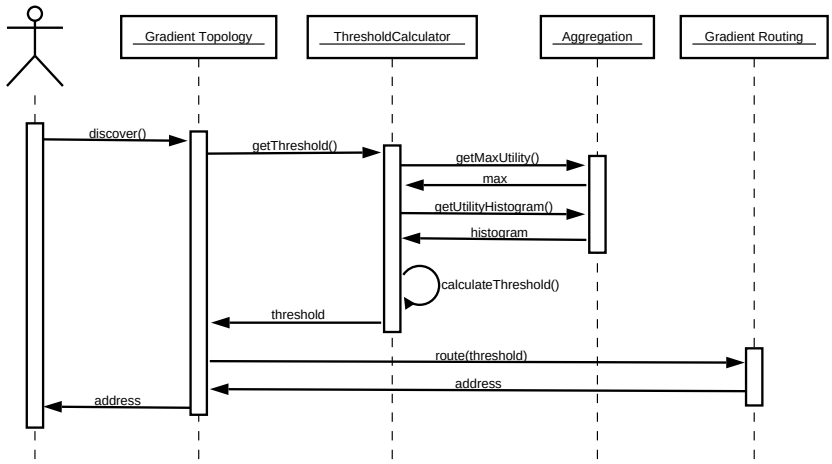

Figure 2.10: Super-peer discovery.

- **Reputation Management**: The gradient topology could be integrated into a decentralised reputation management system in order to isolate peers that are not reputable with regards to providing an essential system service.
- **Collaborative Feedback**: Peers in the gradient topology could collaborate on providing feedback to their own experience with their neighbours.

The presented solutions are likely to be combined for optimal efficiency of identifying hostile or untrustworthy peers.

Each peer periodically compares it own utility with the current replica placement threshold, and potentially performs one of the following actions.

- A peer creates a new KB/SR replica if it is utility is higher than the replica placement threshold and it does not host a replica already. The replica installation is performed by the KB/SR replication algorithm.
- A peer removes its KB/SR replica if the peer's current utility falls below the replica placement threshold. The replica removal is performed according to the KB/SR replication algorithm.

Figure 2.11 shows the super-peer election algorithm. Similarly as in KB/SR replica discovery, the peer calculates a utility threshold for KB/SR replicas, and compares the threshold with its own utility. Peers with utility above the threshold become super-peers, while peers with utility below the threshold become ordinary peers. As mentioned before, the replica transfer and replica synchronisation are handled by higher level protocols described in the DBE KB/SR design document.

In the reminder of this section, we describe the Application Programming Interface (API) of the Gradient Topology component in the P2P architecture. The API consists of three sub-interfaces: Management, Routing, and Aggregation interface.

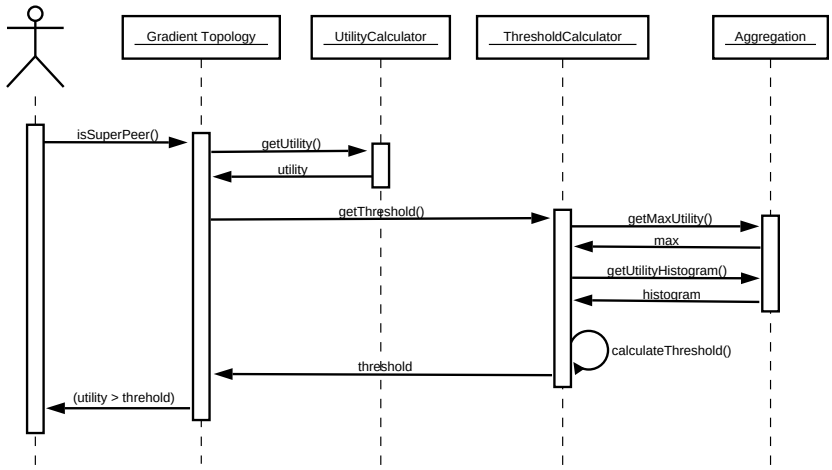

Figure 2.11: Super-peer election.

2.3.1 Management

The Management interface allows the peer to join the gradient topology, leave the topology, and to define its utility and super-peer utility threshold.

- *join()* : This methods allows the peer to join the gradient topology. It requires a list of addresses (at least one address) of peers that are already connected to the gradient topology. The joining peer obtains an initial list of neighbours and initial values of system-wide aggregates from the specified peer(s).
- *leave()* : This method triggers the leave procedure that should be performed by every peer before leaving the gradient topology.
- *setUtilityCalculator()* : This method is used to set the peer's utility function. The `UtilityCalculator` class encapsulates the peer utility function and has only one method, *getUtility()*, that returns the current peer utility.
- *setThresholdCalculator()* : This method allows to set the super-peer utility threshold. The `ThresholdCalculator` class encapsulates the function that calculates the threshold (for example based on the system-wide utility aggregates) and has only one method, *getThreshold()*, that returns the current super-peer utility threshold.
- *isSuperPeer()* : This method returns the current status of the peer, i.e., whether it is a super-peer, determined using the algorithm shown in Figure 2.11.

2.3.2 Routing

The Routing interface enables searching for peers in the system above a given utility threshold. It encapsulates the gradient search algorithm.

- *getPeer()* : This method returns the address of a peer above a given utility threshold. Optional parameters specify the wait timeout and search range (so called

message Time-to-Live, i.e., the maximum number of P2P hops for search messages)

- *getPeers()* : This method is similar to *getPeer()* and returns a set of addresses of peers above given utility threshold. Optional parameters specify the wait timeout, search range (Time-to-Live), and the number of addresses that the method should return.

2.3.3 Aggregation

The Aggregation interface allows peers to obtain estimates of global system properties. The aggregation component continuously maintains the following estimates: the current number of peers in the system, N, the maximum peer utility in the system, Max, and a cumulative histogram of peer utility values, H. The minimum peer utility is defined as zero. The cumulative utility histogram with B bins of width w is defined as

$$H(i) = \Big|\{p \mid U(p) \geq i \cdot w\}\Big|$$

for $i \leq j \leq B$, where $U(p)$ is the utility of peer p. Parameter B is also called the histogram resolution. The histogram is a discrete approximation of the peer utility distribution in B points, where each bin corresponds to a single point of the distribution function.

The Aggregation interface contains the following methods.

- *getNumberOfPeers()* : The method returns the current estimation of the total number of peers in the system.
- *getMaxUtility()* : The method returns the current estimation of the maximum peer utility in the system.
- *getUtilityHistogram()* : This method returns a utility histogram. The histogram is represented as an array, where the i-th field in the array corresponds to the i-th bin in the histogram. Additional method parameters may specify the histogram resolution (number of bins) and the utility range for which the histogram is calculated. By default, the histogram is calculated for all values between the minimum utility (zero) and current estimated maximum utility.

Listing 2.1: Gradient Topology API

```
1  interface MembershipManagement {
2    void join(Address[] initialNeighbours);
3    void setUtilityCalculator(UtilityCalculator uc);
4    void setThresholdCalculator(ThresholdCalculator tc);
5    boolean isSuperPeer();
6    void leave();
7  }
8
9  interface UtilityCalculator {
10   double getUtility();
11 }
12
13 interface ThresholdCalculator {
14   double getThreshold();
15 }
16
17 interface Routing {
18   Address getPeer(double threshold);
19   Address getPeer(double threshold, long timeout);
20   Address getPeer(double threshold, long timoeut, int ttl);
21   Address[] getPeers(double threshold, int count);
22   Address[] getPeers(double threshold, int count, long
         timeout);
23   Address[] getPeers(double threshold, int count, long
         timeout, int ttl);
24 }
25
26 interface Aggregation {
27   long getNumberOfPeers();
28   double getMaxUtility();
29   long[] getUtilityHistogram();
30   long[] getUtilityHistogram(int resolution);
31   long[] getHistogram(double from, double to, int bins);
32 }
```

2.3.4 Multiple Utility Functions

This section describes an approach that allows to build multiple applications with different utility functions on top of the gradient topology.

The goal can be stated as follows. There are two P2P applications, A and B, and each application introduces its peer utility function, U_A and U_B, respectively. Each utility function describes and measures application specific peer requirements. Furthermore, each applications defines its utility threshold, t_A and t_B, accordingly, and the goal for each application is to discover peers with the application-specific utility above the application-specified threshold.

Formally, the utility functions have the following form

$$U_A : P \rightarrow R, \quad U_B : P \rightarrow R$$

where P is the set of all peers in the system, and the two applications require the selection of the following subsets of peers

$$\{p \mid U_A(p) \geq t_A\}, \quad \{p \mid U_B(p) \geq t_B\}$$

There is a number of different approaches to achieve the goal. One approach splits the neighbour sets of all peers into two separate parts, corresponding to application A and B, and generates two separate gradient topologies for both applications. This can be accomplished by applying the neighbour selection algorithm, aggregation algorithm, and gradient search with U_A utility function to the to the neighbour set corresponding to application A, and by applying the algorithms with U_B utility to the neighbour set corresponding to application B. However, this approach does not scale with the number of supported applications since it requires a high number of neighbours per peer, and hence, a high number of exchanged messages per peer.

Another approach combines the application specific utility functions into one general utility function and uses this utility function to generate one gradient topology shared by all applications. Assuming that peer utility is a function of a number of peer parameters that can be directly measured or obtained from the peer, the two utility functions have the following form

$$U_A : R^N \rightarrow R, \quad U_B : R^M \rightarrow R$$

and the combined utility function can be described as

$$U : R^{N+M} \rightarrow R$$

However, utility function composition is a difficult problem. It is not certain how to define the combined utility function, especially if the functions have conflicting requirement for shared parameters, or if no or very few peers satisfy all criteria specified by all applications.

Therefore, we propose an approach, where the combined utility function is defined as

$$U(p) = \max(U_A(p), U_B(p))$$

The above formula specifies that a peer has a high utility if it either has a high U_A or high U_B. sTherefore, if the gradient topology is generated, both high utility peers for application A and high utility peers for application B are located in the core. Gradient search can be used for routing messages to the core and hence for high utility peers

discovery. The only required change in the searching algorithm is that once a message is delivered to a peer in the core, it is not certain what type of high utility this peer has, and therefore, the message may have to be forwarded to a different peer in the core that has the required high utility value. This last step, however, with a very high probability can be achieved in one hop, since peers in the core are well-connected.

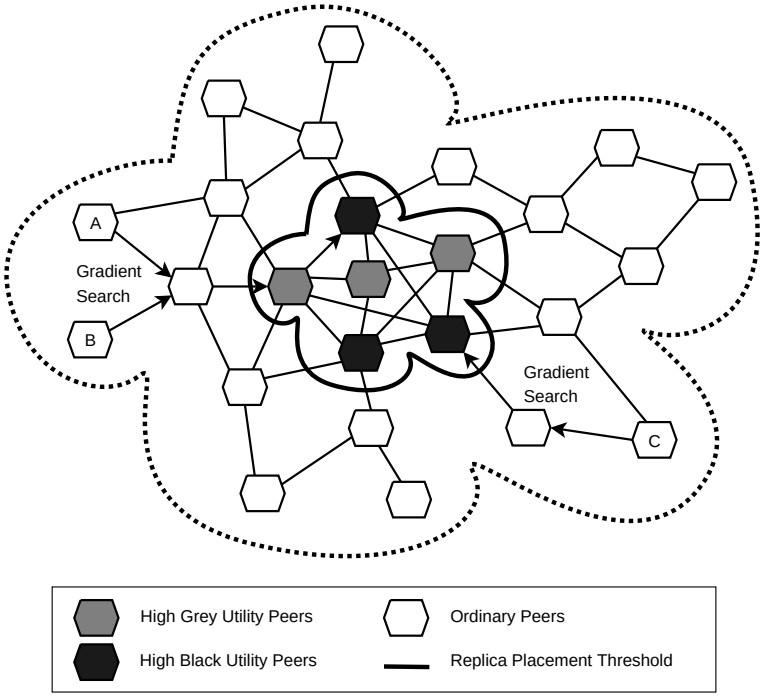

Figure 2.12: High utility peer discovery in a gradient topology with two different utility functions.

Figure 2.12 shows a sample gradient topology supporting two different applications, "grey" and "black", where peers A, B, and C perform gradient search in order to discover "black" high utility peers. Peers A and B locate "grey" peer in the core and their request is forwarded to a "black" high utility neighbour. Peer C locates directly a "black" super-peer in the core.

A potential problem that may appear when using multiple utility functions in one gradient topology is that the utility functions may have significantly different value ranges. I such case, the composed utility U may be dominated by one of the application specific utility functions. For example, if U_A has values within range $[0..1]$ and U_B has values range $[1..100]$, then U, defined as maximum of U_A and U_B, is essentially equal to U_B. In this case, searching for high utility peers for application A becomes inefficient, since U_A is not reflected in utility U.

One solution to this problem is to define all utility functions in such a way that all of them have the same value range, e.g., $[0..1]$. However, this requires system-wide knowledge about peer properties. Simple transformations (projections) of the utility function into a fixed interval, such as the sigmoid functions

$$U_A^{'}(p) = \frac{1}{1 + e^{-U(p)}}$$

or

$$U_A^{'}(p) = 1 - \frac{1}{1 + U_A(p)}$$

do not solve the problem, since such transformations must be monotonic, and hence, if one function has higher values than the other function, the same relation holds when the transformation is applied.

We suggest an approach, where the utility of all peers is scaled by a factor to ensure that different utility functions have comparable values, i.e., within a similar range. The scaling factor can be calculated using the threshold values, t_A and t_B, for example in the following way

$$U_A^{'}(p) = \frac{t_B}{t_A} U_A(p)$$

By doing so, the system ensures that both functions have close utility values for peers near the thresholds, i.e., peers that the applications are searching for.

A similar effect could be achieved by calculating the scaling factor based on the average utilities (U_A and U_B) in the system. We do not recommend using the maximum utility values for the factor calculation, since the maximum utility may change rapidly and hence it can introduce significant instability of the utility function calculation. The dynamism of the utility change can be potentially reduced by applying an adaptive update formula

$$U_A(p) = \alpha U_A(p) + (1 - \alpha) U_A^{'}(p)$$

The utility scaling factor is calculated by each peer that initiates an aggregation round and is propagated between peers together with aggregation messages.

2.4 Distributed Hash Table

In this section we show how we address the problem of service proxy storage, location and removal with Distributed Hash Tables (DHTs).

DBE requires that there is a mapping of unique service identifiers onto service proxies, i.e., given a service identifier (obtained from a Service Manifest), the DBE system should enable users to locate the corresponding service proxy object. This can be achieved by Distributed Hash Tables, which are an active area of recent research [22, 29, 25, 23]. DHTs are peer-to-peer overlay systems that provide a mapping of a large identifier space onto a set of nodes in the system in a deterministic and distributed fashion. This mapping of identifiers to nodes is called *routing* or *lookup*. DHTs generally perform routing using only $O(logN)$ overlay hops in a network of N nodes where every node maintains only $O(logN)$ neighbour links.

The Distributed Hash Table architecture for DBE extends an open-source Bamboo protocol has been chosen [23], which is, in turn, based on Pastry [25]. Bamboo as the underlying protocol for our DHT architecture was due to the following reasons:

- open-source and portable implementation in Java

- designed to handle high churn rates better than other DHTs, which is essential for peer-to-peer environments with many nodes joining and leaving (see [23])
- large community with active mailing-list
- mature and highly tested code used in many projects and a public deployment in the form of OpenDHT [24]
- highly configurable with a support for external modules

In general, Bamboo spreads <key, value> pairs among peers participating in the overlay and provides efficient routing algorithms. It relies on a structured overlay topology, where each peer is assigned a random unique identifier and responsibility for part of the key space. Peers store all pairs whose keys fall in their part of the key space and provide operations such as lookup, remove, insert, which are routed in a multi-hop fashion (i.e. from one peer to another).

Bamboo uses essentially the same algorithm for routing messages as Pastry [25]. Each peer is assigned a 160-bit peer identifier. The identifier is generated randomly when a peer is created and it is assumed that the resulting set of identifiers is uniformly distributed in the identifiers space. For a network of *N* peers Bamboo routes to the numerically closest peer to a given key in less then $log(N)$ steps. The peer's neighbourhood is constructed from peers with identifiers that have the same first *i* digits and differ only at the *i+1* position. When routing, a peer normally forwards the message to a peer whose identifier shares with the key a prefix that is at least one digit longer than the prefix that the key shares with the current peer's identifier. Figure 2.13 shows the example scenario, where the peer whose identifier starts with *111* locates a node closest to identifier *011*. As the network grows in size, each peer on the path can choose among more neighbouring peers that are appropriate for forwarding the packet, hence, it uses additional criteria, such as the estimated route latency, to select the next hop.

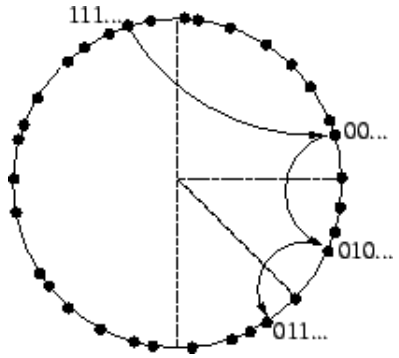

Figure 2.13: To find the node closest to identifier *011*, the node whose identifier starts with *111* sends a lookup message to its neighbour whose first digit is *0*. This node then forwards the query to its neighbour whose first two digits are *01*, and from there the node is forwarded to the neighbour whose first three digits are *011*.

Distributed Hash Tables meet the DBE requirements in that they provide

- *Load balancing*: DHTs spread keys evenly over the participating nodes; this provides a degree of natural load balance.

- *Decentralisation*: DHTs are fully distributed: no node is more important than any other. This improves robustness and makes DHTs appropriate for loosely-organised peer-to-peer applications.
- *Scalability*: The cost of the DHT lookup grows as the logarithm of the number of nodes, so even very large systems are feasible. No parameter tuning is required to achieve this scaling.
- *Availability*: DHTs cope with node joining the system as well as with arbitrary failures; nodes automatically adjust their routing tables .
- *Flexible naming*: DHTs enable flexibility in the selection of keys by users.
- *Fault tolerance*: The values are replicated to a certain number of neighbouring nodes, enabling their recovery if a node responsible for some identifier space unexpectedly fails.

2.4.1 Application of DHT to DBE

The basic DHT functionality essentially meets the use case scenarios presented in this document and enables seamless integration with DBE. In particular, the service ID is a key and the service proxy object is a value in the DHT overlay. Storing a service proxy in the overlay corresponds to the *put* (insert) operation in the DHT, where the node responsible for the service identifier is determined by the DHT routing protocol and the proxy is sent to that node. The proxy lookup corresponds to the *get* (lookup) operation in the DHT, where the node storing the service identifier is determined in the same manner and the proxy object is retrieved from that node. The proxy removal works analogously. Some DHTs such as Bamboo support also a lease mechanism that automatically removes <key, value> pairs after the time determined by the lease in order to prevent overloading nodes with obsolete proxy objects of services that ceased to exist. Thus, service providers need to periodically renew the lease of each of their services by submitting the proxy object again. A node in the DHT that receives a proxy that it already has, stores only one of them and its lease is updated to the maximum of the two leases. Bamboo also introduces a secret key mechanism that relies on user-chosen secret keys being associated with each <key,value> pair. The secret key is then required to remove a service proxy, consequently preventing competing users from removing each others service proxies. However, we have identified several shortcomings of the Bamboo protocol

- support for only one namespace for keys, i.e., there is a possibility of a key conflict if different applications using one DHT overlay let users choose arbitrary keys
- values in Bamboo are limited in size to 1024 bytes, which is not sufficient for storing service proxy objects as well. Other applications using the DHT overlay may require support for larger values (e.g., distributed storage systems)
- even though Bamboo supports secret keys associated with <key,value> pairs, each time a service proxy is updated, a new secret key needs to be generated and remembered by the service provider; this is not convenient as applications need to maintain many different secret keys, move them across machines when the service is migrated and ensure not using them again

We have addressed these shortcoming (see Section 3.2 for details) in an additional layer on top of the Bamboo protocol, thereby maintaining the compatibility with future versions of Bamboo and being able to use PlanetLab deployment of Bamboo if necessary.

In the reminder of this section, we describe how we support NAT'd and firewalled nodes and present the Application Programming Interface (API) of the DHT implementation in the DBE P2P environment. The API consists of two sub-interfaces: Management and Routing.

2.4.2 NAT'd Peers Interactions

A service provider joins the DHT overlay only if it is not behind a Network Address Translation (NAT) or firewall that blocks an outside access to its ports (ports can be specified in a DBE configuration). If the service provider is restricted by firewalls or NATs (see Section 3.3.2 for more details), then it selects a relay peer from the known peers participating in the overlay (using the bootstrap service). The relay peer can be any live node that participates in the DHT overlay. The task of the relay peer is to act as a proxy between a restricted peer and the DHT overlay. Thus, it accepts the requests such as *get*, *put*, *remove*, *updateLease* from the restricted peers, forwards them to the DHT network and possibly returns the replies back the restricted peers (see Figure 2.14 for an example scenario of storing a service proxy in the DHT overlay by a restricted peer). Document refers to nodes that are not restricted and participate in the DHT overlay as *DHT nodes*.

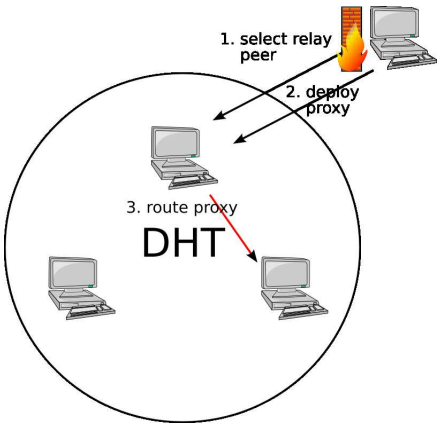

Figure 2.14: An example scenario where a restricted peer stores a service proxy in the DHT via a relay peer

2.4.3 Management

The Management interface (see Listing 2.2) allows the peer to join and leave the DHT overlay. The implementation of this interface reads the necessary configuration value from a properties file.

- *join()* : This methods allows an peer to join the DHT overlay with a randomly generated node identifier. In particular, it requires that the ports specified in the configuration (default ports are 5850-5852) are available and accessible from the

Internet. The method throws an exception if it cannot bind to a port or any other unrecoverable error occurs.

- *leave()* : This method stops all the connections. The node will not accept any routing requests after calling this method.

2.4.4 Routing

The Routing interface (see Listing 2.2) of the DHT enables searching, storing, removing values and updating their leases. In particular, the key can be a service identifier and the value can be a service proxy.

- *put()* : This method enables a node to store a <*key*, *value*, *secret*> tuple in the overlay in the private namespace *appId* for a maximum of *ttl* seconds, after which it will be automatically removed if not updated (with updateLease() function). The private namespace enables a virtual separation of data between different applications that use the DHT. The service proxy storage has a reserved namespace identifier with a value of *1*. The secret key guarantees that the value can be removed only by a user that reveals it (if *null* then anyone can remove the value). The put method returns *0* if the operation finished with success or *-1* otherwise.

- *get()* : This method enables a node to lookup a value based on its unique *key* and a private application namespace (*appId*). It returns up to *maxvals* values (there may be several values stored under one key) in the user provided *values* array (that should be empty when invoking this method). The get method returns *0* if the operation finished with success or *-1* otherwise.

- *remove()* : This method enables a node to remove all values stored under the given *key* in the namespace *appId*. It requires that a secret key (*secret*) is revealed. The method returns *0* if the operation finished with success or *-1* otherwise.

- *updateLease()* : This method enables to update the lease of a value under the given *key* in the namespace *appId* for another *ttl* seconds. Should be called periodically by a service provider for each of its service proxies to ensure that they are available in the overlay. The method returns *0* if the operation finished with success or *-1* otherwise.

Listing 2.2: DHT API

```
1  interface MembershipManagement {
2    void join() throws Exception;
3    void leave();
4  }
5
6  interface Routing {
7    int put(short appId, byte[] key, byte[] value, int ttl,
         byte[] secret);
8    int get(short appId, byte[] key, int maxvals, ArrayList<
         byte[]> values);
9    int remove(short appId, byte[] key, byte[] secret);
10   int updateLease(short appId, byte[] key, int ttl);
11 }
```

2.5 Bootstrap

Bootstrap is a process in which a node obtains an initial configuration in order to join the system. In P2P environments, this involves obtaining addresses of initial neighbours (at least one alive neighbour) through which the joining peer can discover and contact other peers in the system.

The bootstrap mechanism depends heavily on the properties of the underlying network, i.e., the communication media used in the system. For example, in wire-less environments, a node is bootstrapped by broadcasting a "join" or "hello" message over the wireless medium [5, 6] and receiving responses from all nodes in the communication range. However, in wide-area networks, such as the Internet, the broadcast functionality is usually not available. In particular, the IP multicast protocol has not been widely adopted by Internet providers and users due to design and deployment difficulties [7]. Consequently, there is no reliable and fully decentralised bootstrap mechanism for large-scale systems based on the Internet. Existing P2P systems, e.g. [4, 3, 11, 25, 24, 16], commonly rely on centralised solutions, such as centralised *bootstrap nodes*, whose addresses are globally known to all peers. A node that joins the system contacts such bootstrap node(s) and obtains addresses of initial neighbours. The DHT of the DBE SOA is based on the same concept of bootstrap nodes, however, multiple bootstrap nodes may be available to avoid central authorities in a self-organised P2P system.

This mechanism can be further improved by *local caching* of neighbour addresses by peers. In this approach, a peer contacts bootstrap node(s) only once, i.e., upon the first joining the system. For subsequent sessions, a joining peer obtains initial addresses of neighbours from its local cache, unless none of the cached peers are available upon rejoining the network. In this case, the peer would fall back to the bootstrap mechanism.

We have designed and implemented a bootstrap mechanism for the DBE architecture based on the above principles. Detailed description, including the API, is located in Section 3.3.

2.6 Supporting Multiple Topologies

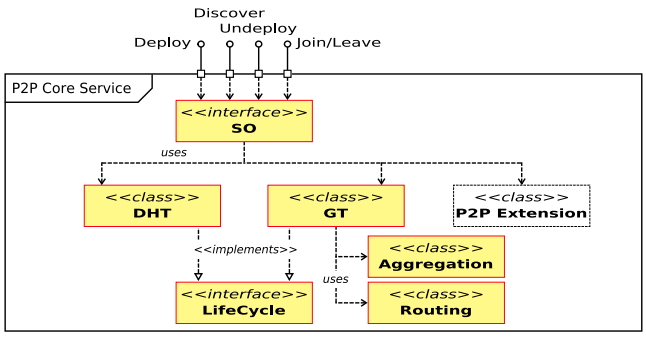

Figure 2.15: The P2P Core Service

In order to account for future developments in the fast growing area of service-oriented computing, it is a necessary requirement to provide an architecture that promotes re-use and allows new components to be fitted in easily. The P2P architecture can be extended as follows:

1. Provide a specialised interface to the DHT topology: The DHT itself provides a generic API in order to retrieve and store values. To meet a specific functional requirement it is good practice to implement a specialised API. In fact, two specialised APIs are provided for the Identity Management and the Proxy repository respectively that share the same DHT topology. Internally, each functional domain has an application identifier assigned to it to avoid any potential storage conflicts. Additional APIs can be developed and integrated in a similar fashion using a different application identifier.

2. Incorporate a highly-specialised topology into the Servent's core-services: The Gradient Topology is a highly specialised topology to meet the requirements for registry placement and provides a heuristic search algorithm to find any registry in the network. Consequently, the routing and aggregation APIs are specific and not suitable for any extensions that are not within the realm of registry placement, gradient organisation, and heuristic search. However, the gradient topology can be extended itself by providing multiple utility levels.

3. Other topologies can be incorporated into the DBE P2P core service. The programming model that is presented in this document provides an abstraction to the SOA principles, deploy, discover, and neighbourhood management. Thus, changes in the underlying P2P topologies do not require client APIs of the DBE P2P service to be modified (see Figure 2.15).

4. The servent component framework simplifies embedding further topologies as core services. This is the suggested procedure for any extensions that require specific solutions to the SOA platform.

To conclude this section, we believe that extension points are provided in order to meet future demands to some degree. Two solutions are provided, whereby one is more suitable for pure storage with the typical characteristics of a DHT overlay, while the other one is more appropriate for specialised topologies.

Chapter 3

Implementation

In this chapter, we present the implementation of the P2P architecture to support a self-organising SOA with a focus on the integration into the DBE Servent. We would like to point out that the architecture in general is a framework that can be used as an extension to any SOA middleware in order to replace centralised components to avoid single points of failure.

This chapter reflects the structure of the design chapter (see Chapter 2) in that it provides algorithms for the specific functionalities to manage, route, and aggregate within each topology. Section 1 deals with the Gradient Topology followed by Section 2 covering the DHT. The last section delineates solutions to the bootstrap problem.

The result of this implementation is part of the open-source project Swallow.

3.1 The Gradient Topology

This section describes in detail the implementation of the gradient topology peer. The description includes the main software components, their functionality, and applied algorithms.

The gradient topology interface (API) and the main three sub-interfaces, i.e., Management, Routing, and Aggregation, described in Section 2.3, are implemented in the following way.

- Aggregation - This interface is entirely implemented by the Aggregation component described in Section 3.1.3.
- Routing - This interface is entirely implemented by the Gradient Search component described in Section 3.1.5.
- Management - The implementation of this interface is split between multiple components.
  - *join* : This method initialises all gradient topology components and provides initial neighbour addresses for the Neighbourhood Management component described in Section 3.1.2.
  - *leave* : This method shuts down all gradient topology components. In particular, it triggers the Aggregation leave procedure, described in Section 3.1.3.

– *setUtilityCalculator* : This method sets the Utility Calculator described in Section 3.1.1.
– *setThresholdCalculator* : This method sets the Threshold Calculator described in Section 3.1.4.
– *isSuperPeer* : This method encapsulates the super-peer election algorithm covered in Section 3.1.4.

Figure 3.1 shows the general overview of the gradient topology implementation. Table 3.1 on page 45 shows a summary of gradient topology components and provides references to individual components' descriptions.

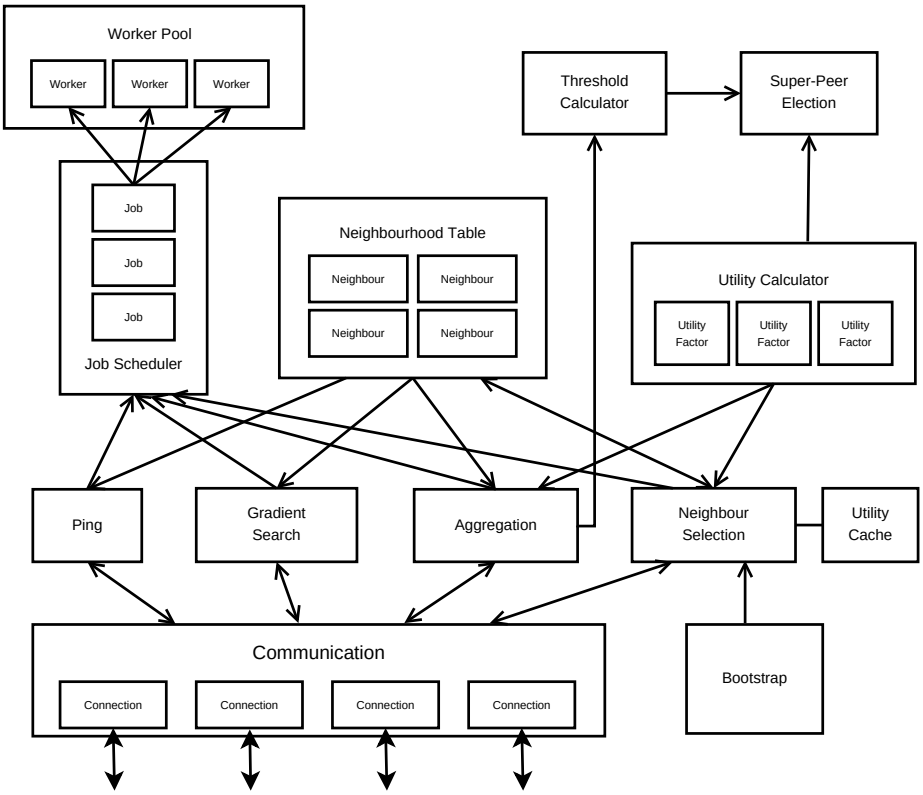

Figure 3.1: Gradient topology implementation.

The remainder of this section describes in detail the implementation of each of the gradient topology components.

3.1.1 Utility Calculator

The Utility Calculator encapsulates the utility function used by the peer. The utility is a metric that measures the capability of the peer to provide services to other peers.

Component	Section
Utility Calculator	3.1.1
Utility Factor	3.1.1
Neighbour Selection	3.1.2
Neighbourhood Table	3.1.2
Utility Cache	3.1.2
Aggregation	3.1.3
Threshold Calculator	3.1.4
Super-Peer Election	3.1.4
Communication	3.1.6
Ping	3.1.6
Job Scheduler	3.1.7
Worker Pool	3.1.7

Table 3.1: Summary of gradient topology components and references to their descriptions.

3.1.1.1 API

Utility Calculator has a very simple interface (see Listing 3.1) with only one method, `getUtility()`, that calculates the current peer's utility value. The minimum utility is zero and the maximum utility is unbounded.

Listing 3.1: Utility Calculator API

```
1 interface UtilityCalculator {
2   double getUtility();
3 }
```

3.1.1.2 Algorithm

The utility function is domain-specific and depends entirely on the application requirements. However, it is expected that many P2P applications will share similar requirements, such as the peer stability or quality of peer network connection.

A very important property of the gradient topology is that the utility function is orthogonal to the neighbour selection algorithm, which generates the topology, and to the searching algorithm. The only assumption these algorithms make about the utility function is that every peer calculates some utility value.

We expect that the utility function can be potentially based on the following peer parameters: peer's availability, current uptime, available storage space, average bandwidth, average latency, CPU performance, and open IP address. In order to estimate the above parameters, the Utility Calculator needs to interact with the Communication Layer (see Section 3.1.6) in order to learn the peer's network connection quality, and with the operating system in order to estimate other peer characteristics, such as the amount of available disk space.

A peer's utility is the main criteria when selecting peers for a service (replica) placement. There are two general approaches for peer utility management.

- Peer utility is calculated automatically by the system (i.e., peer's software). In this case, the system automatically selects peers for service/replica placement.

At any time, a peer may be requested to host a new service/replica, or an existing service/replica may be removed from a peer.

- Peer utility is controlled by the peer's administrator. In this approach, peer administrators can control (through peer utility manipulation) which peers host services/replicas and can decide when a new service/replica is created on or removed from a peer.

The former approach is preferable, since it enables system self-organisation. However, in some systems the latter approach may be required by the system security or administrative policy.

In our implementation, we provide building blocks that calculate *utility factors* (basic peer parameters) that can be used to implement a custom utility function. Currently, we have developed modules that calculate the following utility factors.

- *UptimeCalculator* - that measures a peer's current uptime
- *StorageSpaceCalculator* - that obtains from the system the current amount of available storage space
- *ProcessingSpeedCalculator* - that measures the available amount of CPU processing power
- *LatencyCalculator* - that calculates a peer's average latency in exchanging messages with other peers in the system, using communication statistics obtained from the Communication Layer

Our framework can be easily extended by adding modules that calculate other utility factors. Given N different utility factors, $f_1, f_2, ..., f_N$, the utility function can be defined as

$$U(p) = \alpha_1 f_1 \cdot \alpha_2 f_2 \cdot ... \cdot \alpha_N f_N$$

where $\alpha_1, \alpha_2, \alpha_N$ are constant scaling values. According to this utility definition, a peer has a high utility value if all of its basic utility factors are relatively high, i.e., no factor has an extremely low value. This property can be further strengthen by adding the logarithm

$$U(p) = \alpha_1 \log(f_1) \cdot \alpha_2 \log(f_2) \cdot ... \cdot \alpha_N \log(f_N)$$

In the latter definition, peers do not have a high utility if only one of their utility factors is very high. In order to achieve high utility, a peer needs to have high values of all of its basic utility factors.

3.1.2 Neighbourhood Management

This layer implements the neighbour selection algorithm and is responsible for the gradient topology maintenance. It manages the peer's *neighbourhood set* that stores the address of all current peer's neighbours. The neighbourhood set of a peer p consists of two parts: a *similarity-based* set, S_p, that contains neighbours with similar utility to peer p, and a *random* set, R_p, that contains a nearly random sample of peer addresses in the system. The former set allows to generate the gradient topology structure and enables gradient searching, while the latter set is used by the aggregation

algorithm. Random connections also significantly reduce network partition probability and decrease the network diameter. Apart from neighbourhood sets management, the neighbour selection algorithm maintains estimates of the neighbours' utility values.

3.1.2.1 API

The component provides the following interface for the higher layers (see Listing 3.2).

- *getNeighbours()* : This method returns the set of currently established neighbours.
- *getRandomNeighbours()* : Returns the current random set (R_p).
- *getSimilarNeighbours()* : Returns the current similarity-based set (S_p).
- *getNeighbourUtility()* : Returns the estimated utility value of a given neighbour.
- *setInitialNeighbours()* : Method used in peer initialisation. Sets the initial neighbourhood set.

Listing 3.2: Neighbourhood Management Component API

```
1 interface NeighbourSetManagement {
2   Address[] getNeighbours();
3   Address[] getRandomNeighbours();
4   Address[] getSimilarNeighbours();
5   double getNeighbourUtility(Address neighbour);
6   void setInitialNeighbours(Address[] neighbours);
7 }
```

3.1.2.2 Algorithm

The neighbour selection algorithm performed by each peer in the system is the following. Peers periodically gossip with each other and exchange their neighbourhood sets. On receiving both neighbourhood sets from a neighbour, a gossipping peer selects one entry whose utility level is closest to its own utility and replaces an entry in its similarity-based set. This behaviour clusters peers with similar utility characteristics and generates the gradient topology structure. In addition, a gossipping peer randomly selects an entry from the received random set and replaces a random entry in its random set. Connections to random peers allow peers to explore the network in order to discover other potentially similar neighbours. This significantly reduces the possibility of more than one cluster of high utility peers forming in the network. Random connections also reduce the possibility of the gradient topology partitioning due to excessive clustering. Moreover, random connections between peers are used by the aggregation algorithm described in the Section 3.1.3. Peer p removes a random entry from S_p or R_p if the number of entries in the sets exceeds the maximum allowed number of connections. Figure 3.2 shows the neighbourhood set exchange between two peers.

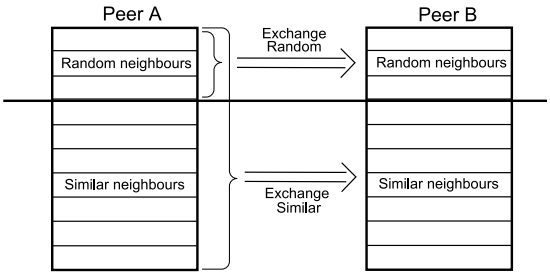

Figure 3.2: Neighbourhood set exchange from Peer A to Peer B.

In addition to the neighbour sets, a peer p maintains a cache U_p that stores an estimated utility value, $U_p(q)$, for each neighbour q. Entries in the cache are timestamped and peers exchange these entries whenever they gossip.

The emergence of a gradient topology is a result of the system's self-organisation. Peers are independent, have limited knowledge about the system and interact with a limited number of neighbours. Utility can be considered as a microscopic property of a peer which enables through peer interaction the construction of the macroscopic gradient structure.

Simulation of the neighbour selection algorithm, described in [26], shows that the algorithm generates a P2P topology with a very small diameter (an order of 5-6 hops for 100,000 peers) and that it has a global gradient structure.

3.1.2.3 Related Work

A number of P2P systems based on super-peers have been proposed. Yang and Molina [34] investigate general principles of designing super-peer-based networks, however, they do not provide any specific super-peer election algorithm. OceanStore [14] proposes to elect a primary tier "consisting of a small number of replicas located in high-bandwidth, high connectivity regions of the network" for the purpose of handling updates, however, no specific algorithm for the election of such a tier is presented. Brocade [35] improves routing efficiency in a DHT by exploiting resource heterogeneity, but unlike our approach, it does not address the super-peer election problem.

Chord [29, 21] shows that the load between peers can be balanced by assigning multiple *virtual servers* to high performance physical hosts. The DHT structure may be used for the discovery of under- or over-loaded peers using *random sampling*, distributed *directories*, and other similar techniques. Mizrak et al [17] proposes the use of high capacity super-peers to improve routing performance in a DHT. However, these systems focus on load balancing in a DHT rather than the selection of potential super-peers from the set of all peers in the system.

Montresor [18] proposes a protocol for super-peer overlay generation, however, unlike our gradient topology, his topology maintains a discrete (binary) distinction between super-peers and client peers. In contrast, our approach introduces a continuous peer utility spectrum and approximates the distribution of peer utility in the system in order to discover peers above an adaptive utility threshold. Our neighbour selection algorithm can be seen as a special case of the T-Man protocol [11] that generates a gradient topology, where the ranking function is based on our peer utility metric. The advantage of such a utility ranking function is that applications can exploit the constructed topology in order to elect appropriate super-peers.

3.1.3 Aggregation

The aggregation component enables an estimation of global peer properties in the system. The component continuously maintains the following estimates: the current number of peers in the system, N, the maximum peer utility in the system, Max, and a *cumulative histogram* of peer utility values, H. The cumulative utility histogram with B bins of width w is defined as

$$H(i) = \left| \{p \mid U(p) \geq i \cdot w\} \right| \tag{3.1}$$

for $i \leq j \leq B$, where $U(p)$ is the utility of peer p. Parameter B is also called the histogram resolution. The histogram is a discrete approximation of the peer utility distribution in B points, where each bin corresponds to a single point of the distribution function.

The estimated system properties provided by the aggregation algorithm (especially the peer utility distribution) allow peers to estimate what constitutes high utility in a running system. This enables the calculation of adaptive thresholds for high utility peers. These high-utility peers, above the selected threshold, may be used by the system for services (or data) placement and maintenance.

3.1.3.1 API

Aggregation component implements the following interface (see Listing 3.3).

- *getNumberOfPeers()* : this method returns the current estimation of the total number of peers in the system.
- *getMaxUtility()* : Returns the current estimation of maximum peer utility in the system.
- *getUtilityHistogram()* : This method returns a utility histogram that approximates the distribution of peer utilities in the system. The histogram is represented as an array, where the i-th field in the array corresponds to the i-th bin in the histogram. The utility range of the histogram is $0..max$, where max is the estimated current maximum utility in the system.
- *getUtilityHistogram(int resolution)* : This method calculates a utility histogram with a given number of bins (see *getUtilityHistogram* method above).
- *getUtilityHistogram(double from, double to, int resolution)* : Creates a utility histogram for a given utility range with a given number of bins (see *getUtilityHistogram* method above).
- *setInitialAggregates()* : Sets the initial estimate values obtained when joining the topology.
- *leave()* : This method should be called by every peer before leaving the system. It ensures the correctness of the aggregation algorithm.

Listing 3.3: Aggregation Component API

```
interface Aggregation {
  long getNumberOfPeers();
  double getMaxUtility();
  long[] getUtilityHistogram();
  long[] getUtilityHistogram(int resolution);
  long[] getUtilityHistogram(double from, double to, int
      resolution);
  void setInitialAggregates(long n, double max, long[]
      histogram);
  void leave();
}
```

3.1.3.2 Algorithm

The aggregation algorithm continuously maintains estimates of global system properties. A peer, p, has an estimate of the current number of peers in the system, N_p, the maximum peer utility in the system, Max_p and a cumulative histogram of peer utility values, H_p. Each of these values approximate the true system properties N^*, Max^*, and H^*.

Peers joining the network contact any peer already in the system and obtain an initial set of neighbours and a current approximation of N^*, Max^*, and H^*. A newly joining peer has minimum utility, which is zero, and the maximum utility of any peer is unbounded. The number of histogram bins, B, is constant in the algorithm.

Peers periodically execute a gossip-based algorithm, where at one step (or round) of the algorithm a peer can send (push) messages to a number of neighbours and receive messages sent by its neighbours in the previous round. A sequence of steps that leads to a new approximation of N^*, Max^*, and H^* is called an aggregation epoch. An epoch can be potentially initiated by any peer at any time step, and the information about the newly created epoch is gradually propagated to all peers in the system. In order to distinguish between different, possibly overlapping, epochs, each epoch is tagged by a unique identifier selected by the initiating peer. Every peer p maintains a cache $cache_p$ that stores the identifiers of aggregation epochs that this peer has participated in. The duration of an epoch is delimited by a time-to-live value. At the end of an epoch, every peer p updates its estimates N_p, Max_p, and H_p.

The algorithm performed at each step by a peer p is shown in Figure 14, and it is based on the algorithms described by Kempe [13] and Montresor [19]. In line 1, peer p starts a new aggregation epoch with probability $\frac{1}{(FN_p)}$. Thus, a new epoch is started by the system with average frequency $\frac{1}{F}$ (every F time steps). The epoch is initiated by creating an aggregation message with a new epoch id and a weight $w = 1$, as specified by Kempe. The ttl field is initialised with an $O(\log(N_p))$ value, since informally speaking, the propagation speed of push-based epidemics is exponential and requires only $O(\log(N_p))$ steps with high probability to reach all N^* peers in the system. The histogram bin width is calculated as $hw = (\frac{Max_p}{B})$.

Furthermore, aggregation messages include a field used to estimate N^* labelled n, a field used to estimate Max^* labelled max, and finally, a histogram, h, consisting of B entries representing individual histogram bins. By combining all aggregation information in one message, the algorithm reduces the total number of messages generated,

and thus limits the network traffic generated. For a 100-bin histogram, the aggregation message size is 842 bytes (see Table 3.2 on page 51).

Field	Description	Size
id	Epoch Identifier	8
ttl	Message Time-To-Live	2
w	Message Weight	8
n	Number of Peers	8
max	Maximum Peer Utility	8
hw	Histogram Width	8
h_1	Histogram Bin 1	8
h_2	Histogram Bin 2	8
...	...	...
h_B	Histogram Bin B	8

Table 3.2: Aggregation message format and size.

In lines 2-12 of Figure 3.3, a peer performs the aggregation of received messages. A peer that receives an aggregation message with a new epoch identifier, i.e., with an id field that is not stored in the cache, joins this new aggregation (lines 13-20) by adding the value of 1 to its n field and to all histogram bins according to Equation 3.1. If the ttl value is less than 1 (indicating the end of the epoch), a peer updates its current estimates of the system properties (lines 21-25). Otherwise, the peer emits a message to a random neighbour (obtained from the neighbour selection algorithm) and to itself so that this peer will continue to participate during the next aggregation round (lines 26-30).

The algorithm exhibits the property of mass conservation defined by [13] provided that no peers fail during an aggregation epoch. At any time step, for each aggregation epoch, the sum of the weights of all aggregation messages in the system is always equal to one, i.e., $\sum_{i=1}^{N} w_i = 1$. Furthermore, the sum of n fields of all messages is equal to the number of peers participating in the aggregation, the maximum of max fields is equal to the maximum utility among peers participating in the aggregation, the average value of ttl fields of all messages at subsequent rounds decreases by one, and for $i \leq j \leq B$, $\sum_{i=1}^{N} h_i(j) = H^*(j)$.

In order to ensure mass conservation, each peer leaving the system is required to perform a leave procedure shown in Figure 3.4. In lines 1-11 of this figure, a peer aggregates currently buffered messages (as in lines 2-12 of Figure 3.3). In lines 12-15 of Figure 3.4, the peer subtracts the value of 1 from the n field and from the histogram bins. Finally, in lines 16-18, the peer sends a message containing the aggregated values to a random neighbour. Section 4.1.2 and Section 4.1.3 present the evaluation results that show a robust network even if a fairly large fraction of nodes do not cleanly leave the system with the "leave procedure".

During one round of the aggregation algorithm, each peer participating in an epoch generates one aggregation message. The epochs are initiated on average every F rounds (frequency $\frac{1}{F}$), and since each epoch lasts on average TTL rounds, the average number of aggregation messages generated and received by each peer in one round is bounded by $O(TTL/F)$, or $O((\frac{1}{F})\log(N))$ if TTL is $O(\log(N))$.

Evaluation of the aggregation algorithm [28] shows that it scales well to at least 50,000 peers and that it enables live approximation of system properties in the face of high churn rates.

Algorithm 1: Aggregation algorithm at a peer p at round t.

with probability $\frac{1}{F \cdot N_p}$ send message $\left(rand(), TTL, 1, 0, 0, \frac{Max_p}{B}, 0\right)$ to self
forall epoch identifiers id **do**
  let $\{m_i\}_{id}$ be messages received at round $t-1$ with epoch identifier id
  $M \leftarrow |\{m_i\}_{id}|$
  let $(id_i, ttl_i, w_i, n_i, max_i, hw_i, h_i)$ be m_i
  $ttl \leftarrow \sum_{i=1}^{M} \frac{ttl_i}{M}$
  $w \leftarrow \sum_{i=1}^{M} w_i$
  $n \leftarrow \sum_{i=1}^{M} n_i$
  $max \leftarrow$max(max_i)
  for $1 \leq j \leq B$ **do**
    $h(j) \leftarrow \sum_{i=1}^{M} h_i(j)$
  end
  if $id \notin cache_p$ **then**
    $n \leftarrow n+1$
    $max \leftarrow$max$(U(p), max)$
    for $1 \leq j \leq \lfloor \frac{U(p)}{hw} \rfloor$ **do**
      $h(j) \leftarrow h(j)+1$
    end
  end
  $cache_p \leftarrow cache_p \cup \{id\}$
  if $ttl < 1$ **then**
    $N_p \leftarrow \frac{n}{w}$
    $Max_p \leftarrow max$
    $H_p(j) \leftarrow \frac{h(j)}{w}$
  end
  else
    $m \leftarrow (id, ttl-1, \frac{w}{2}, \frac{n}{2}, max, hw, h)$
    send m to a random neighbours and to self
  end
end

Figure 3.3: Aggregation algorithm at peer p at time t.

Algorithm 2: Leave procedure at peer p

forall epoch identifiers id **do**
  let $\{m_i\}_{id}$ be all currently buffered messages with epoch identifier id
  $M \leftarrow |\{m_i\}_{id}|$
  let $(id_i, ttl_i, w_i, n_i, max_i, hw_i, h_i)$ be m_i
  $ttl \leftarrow \sum_{i=1}^{M} \frac{ttl_i}{M}$
  $w \leftarrow \sum_{i=1}^{M} w_i$
  $n \leftarrow \sum_{i=1}^{M} n_i$
  $max \leftarrow$max(max_i)
  for $1 \leq j \leq B$ **do**
    $h(j) \leftarrow \sum_{i=1}^{M} h_i(j)$
  end
  $n \leftarrow n - 1$
  for $1 \leq j \leq \lfloor \frac{U(p)}{hw} \rfloor$ **do**
    $h(j) \leftarrow h(j) - 1$
  end
  $m \leftarrow (id, ttl - 1, w, n, max, hw, h)$
  send m to a random neighbour
end

Figure 3.4: Leave procedure at peer p.

3.1.3.3 Related Work

Kempe et al [13] describes a push-based gossip algorithm for the computations of sums, averages, random samples, and quantiles, and provides a theoretical analysis of the algorithm. Montresor, Jelasity and Babaoglu [19, 12] introduce a push-pull-based approach for aggregate computation, however, their model assumes that a message exchange between any two peers is atomic and that the clocks of peers are synchronised. We have extended Kempe's algorithm to calculate histograms, and we have added a peer leave procedure that improves the behaviour of the algorithm in the face of peer churn. We are using the aggregates for adaptive super-peer threshold calculation.

3.1.4 Threshold Calculator

The Threshold Calculator encapsulates the utility threshold function used by the peer. Peers with their utility above the threshold may be selected by the system to become super-peers.

3.1.4.1 API

The threshold Calculator has one method, *getThreshold()*, (see Listing 3.4) that calculates the current utility threshold for a super-peer.

Listing 3.4: Threshold Calculator API

```
1 interface ThresholdCalculator {
2   double getThreshold();
```

```
3 }
```

3.1.4.2 Algorithm

The threshold function is application specific and can be calculated in an arbitrary way defined by the system designer. In the simplest case, the threshold can be given as a fixed, absolute utility value.

However, in many applications a peer needs to estimate the distribution of peer utility in the system in order to know what constitutes high utility threshold in a running P2P system. Otherwise, if the threshold is static (e.g. hard-wired), it may happen that there is no peer in the system that has its utility higher than the selected threshold (in which case no super-peer can be elected), or that the threshold is very low and hence sub-optimal (since most peers in the system will satisfy the super-peer selection criteria). Moreover, due to a system's dynamism, the threshold has to be continuously adapted to the system's current state and peer availability.

We propose an adaptive approach where each peer estimates the utility distribution in the system using an aggregated utility histogram and calculates a super-peer utility threshold based on this distribution. The cumulative utility histogram H with B bins of width w has been defined in Section 3.1.3 as

$$H(i) \quad = \quad \Big|\{p \mid U(p) \geq i \cdot w\}\Big|$$

for $i \leq j \leq B$, where $U(p)$ is the utility of peer p. The cumulative utility distribution function D is defined as

$$D(u) \quad = \quad \Big|\{p \mid U(p) \geq u\}\Big|$$

The utility histogram approximates the utility distribution in B points, i.e., $H(i) = D(iw)$ for $i \leq j \leq B$. We define the *rank threshold*, as a utility value, t_k, such that K highest utility peers have their utilities above or equal t_k, and all other peers have utilities below t_k. Given the utility distribution function, D, the threshold is described by

$$D(t_k) = K, \quad t_k = D^{-1}(K)$$

Rank threshold, if used for super-peer election, enables a straight-forward control of the number of super-peers in the system. It has a property that regardless of peers joining and leaving, there is always K peers in the system above the threshold.

Rank threshold can be estimated using a B-bin and w-width utility histogram, H, with the following expression

$$t_k \quad \approx \quad w * \max_{1 \leq i \leq B} (H(i) \geq K)$$

Similarly, we define the *proportional threshold*, as a utility value, t_F, such that a fixed fraction F of peers in the system (or percentage P of peers, i.e., a quantile) have their utility values above or equal t_F and all other peers have utilities below t_F. Proportional threshold is described by formula

$$D(t_p) \quad = \quad F * N$$

where N is the number of peers in the system, and it can be approximated using a utility histogram

$$t_k \approx w * \max_{1 \leq i \leq B} (H(i) \geq F * N)$$

Proportional threshold has an advantage over the fixed size threshold that as the system grows or shrinks in size, the number of super-peers is accordingly increased or decreased, so that the ratio of super-peers to ordinary peers remains constant.

The simple strategy, where a peer becomes a super-peer when its utility exceeds the super-peer utility threshold and a peer becomes an ordinary peer when its utility falls below the super-peer threshold, has the drawback that peers with utilities close to the threshold value are likely to frequently change their status (i.e., super-peers or ordinary peer status) as their utility fluctuates. This in turn introduces unnecessary overhead to the system.

In order to avoid such behaviour, two utility thresholds are defined, an *upper threshold*, t_u, and a *lower threshold*, t_l, where $t_u > t_l$. The super-peer election algorithm works in the following way. An ordinary peer becomes a super-peer when its utility exceeds the upper threshold, while a super-peer becomes an ordinary peer when its utility drops below the lower threshold. This way, peers with their utility between the higher and lower thresholds never change their super-peer status. The minimum utility change required for a peer to change its status is given by $\triangle = t_h - t_l$.

3.1.5 Gradient Search

This layer encapsulates a searching algorithm, called gradient search, that enables high utility peers discovery. The goal of the search algorithm is to discover a (super)peer, or a set of (super)peers in the system, that have their utility above a certain threshold. The value of the threshold is assigned by the peer that initiates the search and may be calculated using the utility histogram generated by the algorithm described in Section 3.1.3.

3.1.5.1 API

- *getPeer(double utilityThreshold)* : This method initiates a search for a peer above the given utility threshold and returns the address of a discovered peer (or null if no peer above the threshold has been found).
- *getPeer(double utilityThreshold, int count)* : Performs a search and returns a set of count addresses of peers above given utility threshold.

Listing 3.5: Gradient Search API

```
1 interface GradientSearch {
2   Address getPeer(double utilityThreshold);
3   Address[] getPeers(double utilityThreshold, int count);
4 }
```

3.1.5.2 Algorithm

The goal of the search algorithm is to deliver a search message from any peer in the system to a high utility peer in the core, i.e., to a peer with utility above a given threshold. Once the search message has been delivered, the address of the high utility peer is returned to the sender. The value of the threshold is assigned by the peer that initiates the search and may be calculated using the utility histogram generated by the aggregation algorithm (see also Threshold Calculation in Section 3.1.4). The sender's address and the utility threshold are included in the search message.

A peer below the specified utility threshold forwards search messages to higher utility peers until a peer is found whose utility is above the threshold. Each message is associated with a time-to-live (TTL) value that determines the maximum number of hops the message can be propagated.

In gradient search, each peer p greedily forwards messages to its highest utility neighbour, i.e., to a neighbour q whose utility is equal to

$$max_{x \in S_p \cup R_p} U_p(x)$$

where S_p is p's similarity-based neighbourhood set and R_p is p's random neighbourhood set. Thus, messages are forwarded along the utility gradient, as in hill climbing and other similar techniques. It is important to note that the gradient search strategy is generally applicable only to a gradient topology. It assumes that a higher utility peer is closer to the core in terms of the number of hops than a lower utility peer. The maintenance of the gradient structure introduces extra overhead, however, the cost of topology maintenance is generally constant per peer, since the neighbour selection algorithm is performed periodically.

In order to prevent message looping, we append a list of visited peers to each message, and we add a constraint that messages are never forwarded to already visited peers. In a greedy search strategy, where messages are always forwarded to the highest utility peer, messages may oscillate around peers with a locally maximal utility. Local maxima should not occur in an ideal gradient topology, however, every P2P system is under constant churn and the topology may contain local deviations.

Assuming no changes to the topology, all messages from a peer are routed along the same path to the same destination, for example to the same service or replica. However, as soon the utility of a peer belonging to the path changes, or a peer leaves or joins the system, the connections between peers may change and hence the routing path may change as well. In this sense the algorithm is non-deterministic. Similarly, messages sent by different peers are delivered to potentially different destinations. Due to the randomisation of the neighbour selection algorithm, it is expected that the selection of high utility peers by gradient search is highly random, and hence the system provides fair load balancing.

The algorithm exploits the information contained in the gradient topology to limit the search space to a relatively small subset of peers and to achieve a significantly better search performance than traditional search techniques, such as flooding, which require the communication with potentially all peers in the system. Our simulation of gradient search [27] shows that it outperforms random walking, in terms of both the number of transferred messages (network traffic generation) and number of delivered messages (search success rate).

3.1.5.3 Related Work

Recent research on P2P systems has primarily focused on Distributed Hash Tables [29, 22, 25, 16], where the main goal is to provide efficient routing between any pair of peers. In our approach, we are focusing on searching for peers with particular properties in the system (high utility), and assuming that system services are placed on these peers, we provide a mechanism that allows the efficient discovery and consumption of these services. Furthermore, DHTs assume that peer identifiers are unique and relatively static, uniformly distributed in a key space. In our approach, the utility is dynamic and may follow any distribution with multiple peers potentially having the same utility value.

A number of techniques have been developed for searching in unstructured P2P networks (e.g., [33], [15], and [31]). However, these techniques do not exploit any information contained in the underlying P2P topology, in contrast to our gradient search heuristic that takes advantage of the gradient topology structure to improve the searching performance. Morselli at al [20] proposed a routing algorithm for unstructured P2P networks that is similar to gradient searching, however, they address the problem of routing between any pair of peers rather than searching for reliable peers or services.

3.1.6 Communication

This section describes the Communication layer that provides a low-level mechanism for peers to communicate with each other.

The communication model is based on asynchronous message passing. This design decision results from the fact that in a P2P system, peers interact with a number of different, continuously changing, peers, and information is often forwarded by peers on behalf of each other. A single piece of information exchanged by peers is encapsulated in a single message. Since a peer's neighbourhood is dynamic, messages may arrive from any peer at any time.

The communication layer is the only component that directly sends and receives messages over the network. The centralisation of the communication infrastructure enables the estimation of a peer's network connection properties, such as the average bandwidth or latency.

As the communication layer is shared by all peer's components (e.g. routing component, neighbourhood management, aggregation, etc.), and each component sending and receiving messages is associated with a *protocol identifier*. This ensures that messages sent by different components do not interfere with each other.

Additionally, since communication is asynchronous and messages may arrive at any time, every component that receives messages must register a message handler. The handler is invoked by the communication layer whenever a message for a given component is received.

3.1.6.1 API

The layer exposes the following interface for higher-level components (see Listing 3.6).

- *sendMessage()* : Method sends a message, i.e., a chunk of data, to a peer identified by an address. The implementation may be either blocking (synchronous), in which case an error can be immediately signalled by an exception, or non-blocking (asynchronous). In the latter case, the method returns immediately,

and a callback mechanism may be used to notify the sender about the operation result.

- *registerMessageHandler()* : Registers a message handler for a component identified by a given protocol id. All messages received with the specified protocol id will be passed to the registered handler.
- getOwnAddress() : Returns peer's own address. This is needed by other components, such as the neighbour selection algorithm, for gossipping with other peers.
- *getAverageLatency()* : Returns the estimated average round-trip network latency.

Listing 3.6: Communication Layer API

```
1  interface Communication {
2    void sendMessage(int protocol, Address dest, byte[] data);
3    void registerMessageHandler(int protocol, MessageHandler
         handler);
4    Address getOwnAddress();
5    int getAverageLatency();
6  }
7
8  interface MessageHandler {
9    void receiveMessage(Address source, byte[] data);
10 }
```

3.1.6.2 Algorithm

The communication interface has been designed in such a way, that it does not enforce any particular low-level communication protocol. For example, the communication component can be implemented using plain UNIX sockets, Java RMI, or Web Services with SOAP. For this reason, this section will not describe the details of the implementation and will only outline the main design decisions.

In the case where the communication layer is built on top of *web services* (WS), almost all communication infrastructure is already provided by the WS engine (application server). In particular, the WS server enables asynchronous messages sending and receiving. The role of the P2P communication layer is to multiplex/demultiplex messages sent and received by higher level components (i.e., different application protocols).

In the case of a pure *TCP-based* implementation, the communication component needs to handle TCP *sockets*. There are multiple different approaches for socket management. The simplest approach is to create a TCP socket for every message sent and close the socket as soon as the message is received. This, however, introduces high overhead for TCP connection set up whenever a message is transferred. A potentially better approach is to create and maintain a TCP connection to every neighbour and to keep it alive as long as the neighbour exists. This strategy, however, implies a tight coupling between the neighbour selection algorithm (neighbourhood management) and the communication component, and may require from a peer to maintain a high number of concurrent TCP connections if the neighbourhood set is large. For many peers, the maximum number of neighbours would have to be limited due to resource constraints.

A different approach is to create a limited-size cache of TCP connections and leave the maximum number of neighbours unbounded. Whenever a message is send to a neighbour, an existing connection to the given neighbour is retrieved from the cache or a new connection is created and stored in the cache if no connection to the neighbour exists at the moment. Connections are removed according to the *Least Recently Used* (LRU) strategy in order to maximise socket utilisation. If the maximum number of connections is set equal to the maximum number of neighbours, this strategy behaves as the previous described approach.

In the case of *RMI*, or *RPC-based* communication protocols, the communication component design is similar to the TCP-based component. However, instead of managing TCP sockets, the communication layer needs to manage stubs bound to remote objects. A similar approach based on a LRU-cache may be used for storing the *stubs*.

In order to detect non-responsive neighbours (for example due to a failure or a silent neighbour departure), the communication component runs periodic *ping* protocol. At each step of the protocol, a random neighbour is selected and the communication layer sends a ping message to this neighbour. If the selected neighbour is alive, its communication layer responds with a pong message indicating the neighbour is alive. Otherwise, a communication error is generated and the neighbour is removed from the peer's neighbourhood table. This simple protocol serves two purposes. First, as already mentioned, it enables the removal of stale entries in the neighbourhood table. Second, it allows continuous calculation of peer's up-to-date latency statistics.

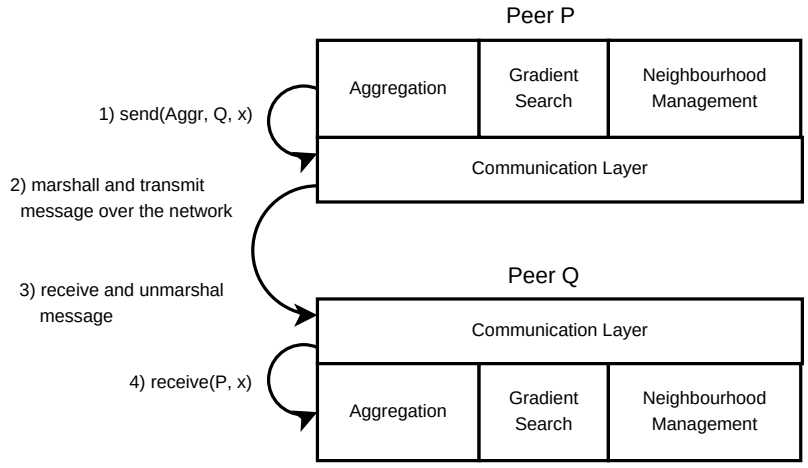

Figure 3.5: Message transfer from Peer P to Peer Q.

Figure 3.5 shows a sample communication scenario, where an aggregation message is sent from Peer P to Peer Q. At step 1, the aggregation component invokes *send()* method on the communication layer, specifying the protocol id (aggregation), destination address (Q, obtained from the neighbourhood management component), and some message content (x). At step 2, the communication layer creates a message, marshals it (converts to a byte sequence), and transfers it to Peer Q. At step 3, the communication component of Peer Q receives and unmarshalls the message. Finally, at step 4, the communication layer of Peer Q calls *receive()* method on the aggregation protocol handler

(aggregation component) supplying the sender's address (P) and message content (x) as method parameters.

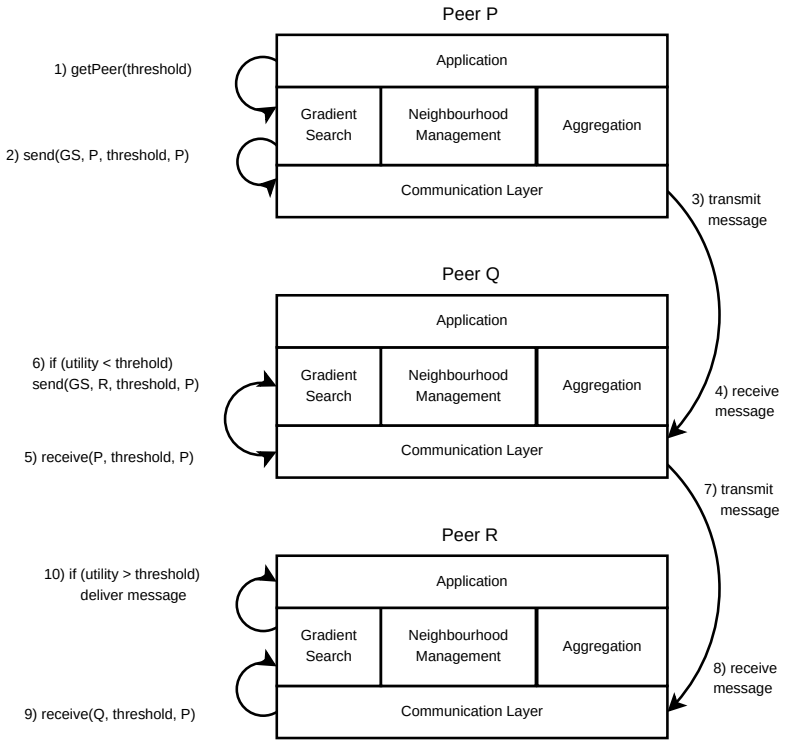

Figure 3.6: Multi-hop routing (gradient search) from Peer P to Peer R.

Figure 3.6 shows a sample scenario of multi-hop routing (gradient search) from Peer P to Peer R through Peer Q. (1) Peer P calculates a *threshold* (potentially using information obtained from the aggregation component) and invokes *getPeer()* method on the gradient search component. (2) Gradient search invokes *send()* method on the communication layer, specifying the protocol id (gradient search), destination address (the highest utility neighbour, Q, obtained from the neighbourhood management component), and the threshold and its own address as the message body. (3) Message is sent to Peer Q. (4) Message is received by Peer Q. (5) Q's communication component invokes *receive()* method on the gradient search module. (6) Gradient search component obtains peer's *utility* from the utility calculator, determines that the peer's utility is lower than the threshold specified in the message, and forwards the message to the highest utility neighbour (Peer R) by calling *send()* method on the communication layer, forwarding message body that contains the utility threshold and original sender P. (7) The messages is transferred to Peer R, (8) and received by R's communication layer. (9) *receive()* method is called on R's gradient search handler. (10) The gradient search component verifies that R's utility is higher than the threshold specified in the message and delivers the message to the application. A return message is generated, containing R's address, and is sent back to the initial request sender (Peer P). Note that this last step is not shown in the Figure 3.6 for clarity reasons.

3.1.7 Concurrency Control

One of the challenges in implementing P2P applications is the high number of concurrent tasks performed by each peer. This high level of concurrency stems from the the fact that every peer simultaneously and asynchronously communicates with a number of dynamically changing neighbours and performs a number of independent (overlapping in time) algorithms, such as routing, aggregation, neighbour selection, etc.

This section describes how we address the problem of concurrency control. In particular, the section shows how we synchronise multiple threads to avoid so called race conditions and deadlocks, and how we are improving the performance by controlling the number of threads created and destroyed in the systems.

We divide all peer activities into *jobs*. A job is a task that can be performed by a single thread. Examples of jobs are sending a message, receiving a message, calculation of peer utility, etc. Each peer maintains a fixed-size pool of *worker threads* that execute jobs. A *job queue* is used to distribute jobs in a fair and thread-safe way between the workers. Furthermore, a *job scheduler* is used for scheduling jobs for execution in the future. Figure 3.7 shows the high-level design of the concurrency control model.

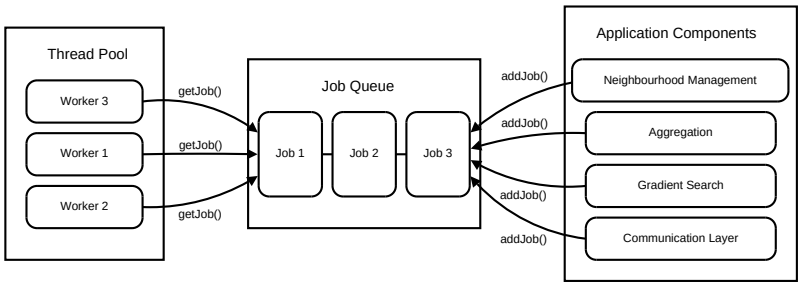

Figure 3.7: Concurrency control model.

3.1.7.1 API

The API consists of a number of interfaces (see Listing 3.7).

- *Job* interface must be implemented by every job object and has only one method, *doJob()*, specialised for every job type, that executes the job.
- *JobQueue* is the object that assigns jobs to worker threads. It has two methods.
  - *addJob(Job job)* : This method adds a new job for execution. It is used by components that generate new jobs, such as neighbour selection algorithm, aggregation, gradient search, etc. When a job is added to the job queue, it is executed as soon as a free worker is available.
  - *getJob()* : This is a blocking method that returns a job pending for execution. It is invoked by the worker threads. If a job is waiting in the queue, it returns immediately. Otherwise, it suspends the worker thread until a job is available.

- *JobScheduler* allows scheduling jobs for execution at a given time. It has one method, *scheduleJob(Job job, long time)*, that specifies a job and a scheduled execution time.
- *ThreadPool* object manages the collection of worker threads. It has two methods, *addWorker(WorkerThread worker)* and *removeWorker(WorkerThread worker)*, that allow creating and removing worker threads.

Listing 3.7: Thread Control API

```
1  interface Job {
2    void doJob();
3  }
4
5  interface JobQueue {
6    void addJob(Job job);
7    Job getJob();
8  }
9
10 interface JobScheduler {
11   void scheduleJob(Job job, long time);
12 }
13
14 interface ThreadPool {
15   void addWorker(WorkerThread worker);
16   void removeWorker(WorkerThread worker);
17 }
```

3.1.7.2 Algorithm

As mentioned above, all peer's activities are implemented as collections of jobs where a job is a task that can be performed by a single thread. Jobs can be created by any peer component. A running job can create new jobs. The following list summarises the most relevant jobs created by gradient topology peers.

- **Neighbour selection** : This job represents one round of the neighbour selection algorithm. The job is repeated periodically. The worker thread that executes this job selects one neighbour and performs one round of gossipping with this neighbour. Details of the algorithm are specified in Section 3.1.2.
- **Aggregation** : Periodically executed job that represents one step of the aggregation algorithm described in Section 3.1.3.
- **Gradient search** : Job created on demand whenever the user (or higher level application) requests a new gradient search or when the peer receives a gradient search message from the network. The job consists of routing the message, i.e., selecting the next hop and forwarding the message. Details of the routing algorithm are covered in Section 3.1.5.
- **Network communication** : This category includes all jobs created by the communication layer in order to handle network communication (see Section 3.1.6). For TCP/IP-based network communication, the following jobs are created.

  - Listening for incoming TCP connection on a server socket
  - Listening for incoming messages on each open connection (i.e., socket)
  - Setting up new TCP connections if needed
  - Sending messages through existing connections (on demand)

- **Periodic Ping** : Periodic job that consists of sending a ping message to a random neighbour. Ping protocol is used by the communication layer for failure detection and network latency measurement (see Section 3.1.6).
- **Utility calculation** : This job calculates current peer's utility value (Section 3.1.1). The utility is used by all major algorithms, such as the neighbour selection, aggregation, and gradient search.
- **Threshold calculation** : The peer periodically calculates the super-peer utility threshold and compares it with its own utility in order to determine its status, i.e., whether it is a super-peer, as described in Section 3.1.4.
- **Job scheduler** : The job scheduler generates a job that blocks one worker thread and waits for the next scheduled job execution.

All jobs that are set for immediate execution are inserted into the job queue. Workers constantly obtain jobs from the queue and execute them. The job queue is a synchronised buffer that allows safe multi-thread access. Insert operations never block. Removal operations block the current thread until an element (i.e., a job) is available in the buffer, as in the producer-consumer design pattern. Jobs are inserted and removed in the FIFO order, which guarantees fairness of job handling by workers and prevents job starvation (i.e., no job stays forever in the buffer).

Additionally, jobs can be scheduled for execution at a time in the future using the job scheduler. The scheduler maintains a list of scheduled jobs together with their execution times. It generates a job that blocks a worker thread and waits until the next scheduled job execution time. When the thread is woken up, the scheduler moves the scheduled job to the job queue (i.e., sets this job for execution) and generates a new job that waits for the next scheduled job execution.

All periodic jobs are scheduled for execution using the job scheduler. This allows limiting the number of threads waiting for periodic job executions to one.

The thread pool contains a collection of running worker threads. The threads are created and started at the peer initialisation time. No new threads are created after the peer initialisation, and idle threads are not destroyed but instead are suspended until jobs are available in the job queue. This solution allows the limitation of the total number of threads running in the system and thus avoids overloading the system when running too many threads at the same time. Furthermore, the system performance is improved by the fact that no threads are created or destroyed at runtime, since the thread pool remains constant.

Each worker thread performs the algorithm shown in Listing 3.8. The worker continuously tries to obtain a job from the job queue, waiting idle if no jobs are available, and executes obtained jobs.

Listing 3.8: Worker Thread Algorithm

```
1 Worker.work() {
2   forever() {
3     // Wait for a job in the job queue
4     Job job = JobQueue.getJob();
5     // Do the job
6     job.doJob();
7   }
8 }
```

Finally, last but not least, a very important problem that arises in multi-threaded application is thread synchronisation. Threads must be synchronised when accessing common resources, such as shared variables, files, TCP/IP sockets, etc. Uncontrolled access may lead to unpredicted system behaviour and is known as the *race condition*.

A common way of synchronising access to shared resources is called *mutual exclusion*, where only one thread at a time can execute a fragment of code called *critical section*. A typical way of achieving this in Java is through the usage of *synchronized* methods. Java objects with synchronized methods behave as *monitors* [8, 9]. However, the usage of monitors introduces the risk of *deadlock*s, i.e., situations where all threads are blocked waiting for access to a monitor. Such a deadlock may occur if a thread that currently occupies a monitor, blocking it from all other threads, is blocked itself waiting for access to another monitor. This is possible if the invocations between monitors (i.e., invocations from a method in a monitor to a method in a different monitor) follow a cycle.

A simple solution to this problem is to structure the code in such a way that no cyclic invocations between monitors are made. However, this imposes strict constraints on the code and is difficult in practise since writing thread-safe, complex programs is known to be very hard for programmers.

In our implementation, we follow another rule. *There is only one monitor in the program*. This single monitor is used to protect all critical sections in the code, such as access to shared variables, sockets, files, etc. This approach eliminates the risk of deadlocks and guarantees the correctness of program execution by multiple threads, while keeping the design and structure of the code simple, and hence, reducing the risk of bugs in the program.

3.2 Distributed Hash Table

This section describes the implementation of the Distributed Hash Table architecture for the DBE system. The DHT architecture is outlined in Figure 3.8 and consists of the bootstrap service (described in Section 2.5), Bamboo DHT protocol (outlined in Section 2.4, see [23] for details), RPC client-server communication layer (Section 3.2.1) and extensions to the Bamboo protocol to support our DHT API (Section 3.2.2, Section 3.2.3 and Section 3.2.4).

3.2.1 Communication Layer

As depicted in Figure 3.8, all DHT operation such as get, put, remove or updateLease that are requested by a *client process* (upper half of the figure) are converted to Remote

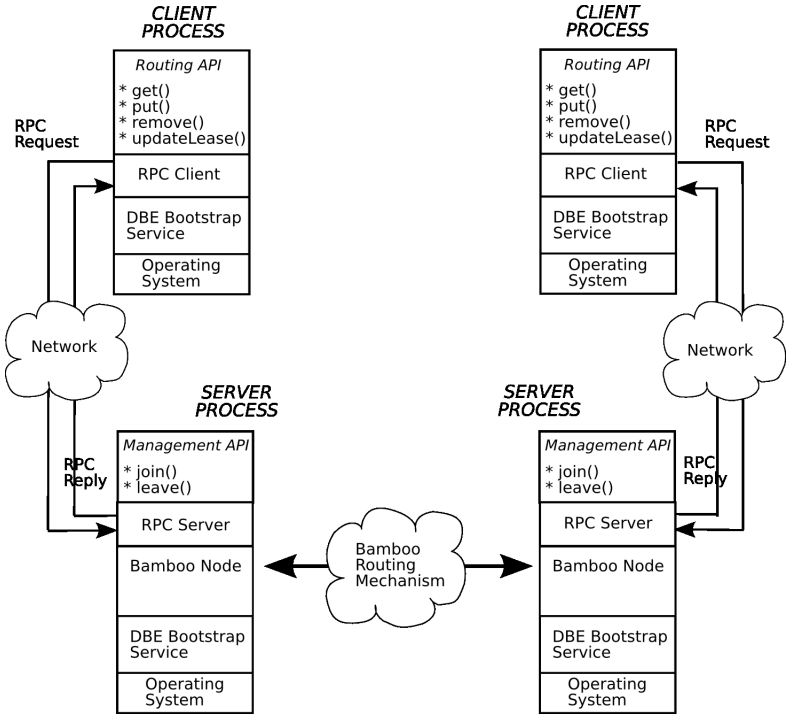

Figure 3.8: DHT Architecture

Procedure Calls (RPCs) and sent to a *server process* (bottom half of the figure). The server process translates them to Bamboo operations and requests the Bamboo layer to execute them. The execution at the Bamboo layer involves the use of the Bamboo routing mechanism. The result of the Bamboo operation (e.g., values stored under a requested key, in the case of the get operation) is then translated again into RPC and sent back to the client process.

The server process provides the Management API (see Section 2.4.3) with methods to join and leave the DHT overlay. The join operation is equivalent to starting a Bamboo node with a custom configuration that depends on the bootstrap service (e.g., custom gateway peers to join the DHT overlay). The leave operation stops the Bamboo node. The client process provides the Routing API (see Section 2.4.4) with methods to get, put, remove and updateLease of data in the DHT. These methods extend the Bamboo methods to support multiple disjoint namespaces, large values and private keys that is described in the following subsections.

The separation of the client process from the server process enables seamless support for restricted peers (NAT'd or firewalled) that need to communicate with the DHT via relay peers (see Section 2.4.2). Service providers that are not restricted by firewalls or NATs deploy both the server and the client process. Thus, their DHT requests (get/put/remove/updateLease) are sent to a local server process that forwards them further into the DHT. In turn, a restricted peer deploys only the client process and sends its requests over RPC to a server process on a selected relay peer that uses Bamboo to fulfil the requests. The result is returned back to the client process on the restricted peer via the same RPC connection.

We assume that a peer can participate in the DBE system if at least it can commu-

nicate with other peers that are not restricted by NATs or firewalls.

3.2.2 Extending DHT to support multiple private namespaces

In Section 2.4 we showed the issues with Bamboo that we had to address in order to adapt it to the DBE requirements. One of these issues was that all applications use one DHT and keys from different applications may conflict with each other. For instance, if two independent applications stored some data under the same key, then they would have a problem with recognising which data belongs to them. This is not a problem if applications generated keys randomly since the key space in the DHT is large (2^{160} different keys). However, the conflict is possible if applications enable users to select arbitrary keys. Thus, we associate each data value with a namespace identifier that each application selects on its own (we assume they are globally known). The *put* operation stores the namespace identifier in the first two bytes of each data value. Each of *get*, *remove* and *updateLease* operations take as an argument a namespace identifier and respectively return, delete or update data only in the selected namespace.

3.2.3 Extending DHT to support values of arbitrary size

Another shortcoming of Bamboo that we identified is a limitation on the size of data values that are stored in the DHT. Bamboo limits their size to only 1024 bytes, which is not enough for most applications including the service proxy storage. We have solved this by implementing an additional layer over Bamboo that handles data values of arbitrary size. The basic idea is to split the large data into smaller blocks and built a Merkle tree over them, which is shown in Figure 3.9. The leaf nodes of the tree contain the data blocks, while inner nodes contain indexing blocks that are composed of pointers to their child nodes. The put operation involves building a tree and saving it in the DHT. The index blocks are composed of pointers that are 20-bytes hash codes of the child block content (SHA-1 is used). Each block that is saved in the DHT has a following structure:

- 2 bytes for the namespace identifier,
- 2 bytes for the inversed level in the tree (0 for a leaf data block and increasing towards the root)
- 1020 bytes for the list of indices to children (20 bytes each) or data, respectively if it is an index block or a data block.

Each block (index or data) in the tree, except for the root block, is stored under a key that is determined by a hash code of the block's content. The root block is stored under the key given as an argument to the put operation.

We provide efficient implementation of put, get, remove and updateLease methods that operate on these Merkle trees. The efficiency of these methods is achieved by parallelising the process of building the tree and storing or retrieving index and data blocks from the Distributed Hash Table. For this purpose, we use a pool of threads that are responsible for sending the blocks over RPC to the server process that, in turn, for each RPC request performs a concurrent DHT routing operation. Once a thread in the thread pool finished some operation, it can be reused for performing other tasks. Thus, our application works in a producer-consumer fashion, where the main thread acts as a

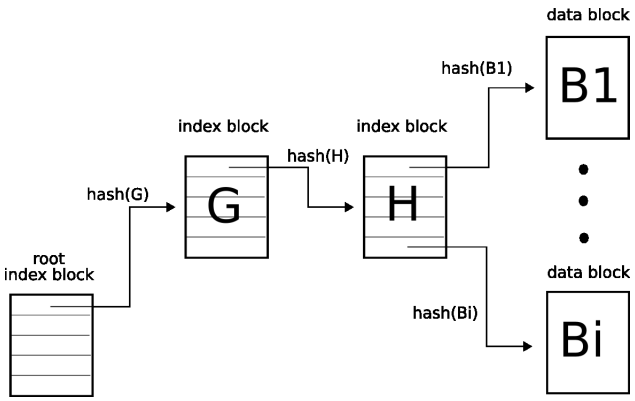

Figure 3.9: Merkle tree constructed for large data values

producer and builds a logical Merkle tree and provides blocks to be sent by consumer threads. The pseudo algorithm for the put operation is depicted in Listing 3.9.

Listing 3.9: Merkle-tree put operation

```
1  int put(namespace, rootKey, largeValue) {
2    blocks = new List();
3    for (block : largeValue split into 1020-bytes blocks) {
4      block.level = 0;
5      block.namespace = namespace;
6      blocks.addLast(block);
7    }
8    while (blocks.size()>1) {
9      indexBlock = new byte[1024]
10     int level = blocks.first().level;
11     indexBlock.level = level+1;
12     indexBlock.namespace = namespace;
13
14     while ((indexBlock is not full) && (blocks.size()>0)
15             && (blocks.first().level==level)) {
16         block = blocks.removeFirst();
17         hash = hashCode(block);
18         do in parallel { RPC_PUT(hash, block); }
19         indexBlock.add(hash);
20     }
21
22     blocks.addLast(indexBlock);
23   }
24
25   //there is now only one root index block left on the list
26   RPC_PUT(rootKey, blocks.first());
27 }
```

In order to account for high churn rates, Bamboo provides a configurable parameter to set the replication strategy for data times that are stored in the DHT to avoid data losses.

The get operation needs to efficiently reconstruct a Merkle-tree given just a key of the root index block, see Listing 3.10. It first retrieves the root index block, then requests its child index blocks and whenever some block arrives it requests its child blocks unless it receives a data block, which is characterised by the level field set to zero. The function parallelises the download of blocks by using a pool of threads. Since the blocks may arrive out of order, priority queues are used to merge them in the correct order, producing the large data value. It is important to note that requesting a large value while the node owning the value updates it does not lead into a race condition with corrupted blocks, because each index block is hash-coded (SHA-1) based on its children. Therefore, it is unlikely that some updates to the tree overtake the process of obtaining the values. However, a large value obtained from the DHT might be invalid by the time the node has merged the individual blocks into the large value, because the node owning the value could have updated the value in the meantime. It is the responsibility of the application layer on top of the DHT to mask these kind of failures.

Listing 3.10: Merkle-tree get operation

```
1  int get(namespace, rootKey) {
2    resultQueue = new PriorityQueue<priority,block>();
3    blocksQueue = new BlockingPriorityQueue<position,block>();
4    rootBlock = RPC_GET(rootKey);
5    blocksQueue.put(<0, rootBlock>);
6    requested = 1;
7
8    while (requested>0) {
9      //gets the first fetched block
10     //or waits if no blocks are available yet
11     <position, block> = blocksQueue.take();
12
13     if (block.level==0) {
14       //it's a data block
15       resultQueue.put(position, block.data);
16     } else {
17       //it's an index block
18       childPosition = TREE_BRANCHING * position;
19       for (int i=0; i < block.indices; i++) {
20         do in parallel {
21           blocksQueue.put(<childPosition,
22                            RPC_GET(block.index[i])>);
23           requested--;
24         }
25         childPosition++;
26         requested++;
27       }
28     }
29   }
30
31   return (resultQueue blocks merged together in the order of
       growing priorities);
32 }
```

The remove operation first uses the above get operation to retrieve the large data value stored under the requested key. Subsequently, it operates in a similar manner to

the described put operation in that a Merkle-tree is built for the data value and blocks are concurrently removed by the pool of threads in a producer-consumer fashion. The updateLease is analogous in that the tree is built and for each index and data block the lease is updated by concurrent threads.

3.2.4 Extending DHT to support a single secret key

We mentioned in Section 2.4 that Bamboo supports *secret keys*, which hash codes (SHA-1 hash) are associated with each <key, value> pair in the overlay. Removing a data requires that the key, value and the full secret key are provided. The key, value and secret key are compared to the ones stored in the overlay and if they match, the data is removed from the overlay. However, the replication scheme employed in Bamboo, where neighbouring nodes store each other replicas, requires that remove operations have associated time-to-live (ttl) values and are stored in the overlay until they expire. This is to avoid scenarios where a node misses a remove operation (e.g., because it was temporarily unavailable) and stores replicas that should have been removed. In Bamboo, the remove operations are stored in the overlay and the node eventually finds out about the remove operation. However, this scheme requires that all subsequent put operations differ in any of the fields: key, value or secret key, or otherwise they will be removed by the previously stored remove operations. For instance, in a particular situation when a user has removed some data and wants to store it again, it needs to produce a new secret key. This is not convenient for applications as it requires maintaining many different secret keys and ensuring that they are not used again. Therefore, we have implemented a layer over the Bamboo protocol that enables applications to always use the same secret key, which we suggest to be the node's private key.

Our extension to the Bamboo protocol changes the way the secret keys are generated, i.e., it ensures that each put operation uses a different secret key, avoiding the aforementioned problems. In order to enable a single *privateKey* to be always used for securing data values, the original Bamboo operation *put(key, value, uniqueSecretKeyHash)* is changed in the following steps:

1. *token* = generate a random sequence
2. *secretKey* = hashCode(*token* + *privateKey*)
3. put(hashCode(*secretKey*), *token*, hashCode(*secretKey*))
4. put(*key*, *value*, hashCode(*secretKey*))

In the first step, a unique random *token* is generated, which is combined in the next step with the local *privateKey* and the hash code of which produces a *secretKey* unique for each put operation. The *token* is stored in the DHT under the key equal to the hash code of the secret key. This is required to later recreate the secret key if the value needs to be removed. Finally, the value is stored under the given key using the hash code of the secret key (the put operation takes as an argument the hash code of a secret key, whereas the get operation requires the full secret key).

The following steps are required to recreate the *secretKey* using the local *privateKey* and consequently remove the *value* under a given *key* from the DHT overlay:

1. <*key, value, secretKeyHash*> = get(*key*)
2. <*secretKeyHash, token, secretKeyHash*> = get(*secretKeyHash*)

3. *secretKey* = hashCode(*token* + *privateKey*)
4. remove(*key*, *value*, *secretKey*)
5. remove(*secretKeyHash*, *token*, *secretKey*)

In the first step, the hash code of the secret key (*secretKeyHash*) that is associated with the data under the requested *key* needs to be obtained. This secret key hash is used in the second step to retrieve the original token from the overlay. The hash code of the combination of the token and the private key produces the original secret key, which is then used to remove the data stored under the requested key. Finally, in the last step the token is removed from the overlay as the same secret key will not be used anymore.

3.2.5 Summary

In this section we described the DHT architecture for the DBE. We showed how NAT'd and firewalled peers are supported through the separation between the client and server processes and the use of relay peers. We presented three extensions to the Bamboo protocol that combined form an abstraction layer over the Bamboo protocol, providing the Management and Routing API discussed in Section 2.4.

3.3 Bootstrap

The bootstrap component, called also *bootstrap service*, allows peers to obtain initial configuration data needed to join the system. This configuration data includes addresses of initial neighbours, and addresses of relay peers. The latter is needed by peers with restricted access to the Internet, i.e., communicating through firewalls or routers with Network Address Translation (NAT). When a peer connects to at least one alive neighbour that is already in the system, potentially using a relay peer, it can receive from this neighbour other initialisation data, such as the addresses of other peers in the system and the current values of aggregates (i.e., number of peers in the system, maximum utility, and the utility histogram).

This bootstrap service is used by both, GT and DHT.

3.3.1 API

The bootstrap service exposes the following interface (see also Listing 3.11).

- *getPeers()* : Returns a list containing addresses of peers that can be used as initial neighbours.
- *getRelay()* : Returns an address of a peer that can be used as a relay peer.
- *changeRelay()* : This method is used by peers to notify the bootstrap service that a relay peer failed and should not be used any more.
- *addPeer()* : This method is used by peers to inform the bootstrap service that a new peer has been discovered that can be used potentially as a relay peer.

Listing 3.11: Bootstrap service API

```
1 interface Bootstrap {
2   Address[] getPeers();
3   Address getRelay();
4   void changeRelay(Address oldRelay);
5   void addPeer(Address newPeer);
6 }
```

3.3.2 Algorithm

The bootstrap procedure consists of two steps, where the second step is performed only if the first step fails.

- Obtain addresses of initial neighbours and relay peers from a *local cache* (stored on a local hard disk) that we call a *local bootstrap service*. A peer leaving the system saves the addresses of its highest utility neighbours in the cache. These addresses are used when the peer is started next time. Note, that this step always fails on a node's first attempt to join the network, because the node has an empty local cache.
- Obtain addresses of initial neighbours and relay peers from a pool of *bootstrap nodes*. The addresses of bootstrap nodes are obtained by resolving globally know DNS name(s). The DNS names may be handled by multiple name servers for reliability reasons. Each DNS name may be resolved to a number of IP addresses belonging to different bootstrap nodes. Round-robin strategy may be used by the DNS servers for the selection of bootstrap nodes. Each bootstrap node maintains a cache of peer addresses that is used to handle bootstrap request from peers joining the system.

A peer contacts bootstrap nodes only if the local cache fails to provide good (i.e. alive and reachable) neighbours and relay peers (if needed). The local cache containing peer addresses is organised in FIFO order – oldest entries are removed when fresh addresses are added. Furthermore, addresses of peers that cannot be contacted are removed.

Every bootstrap node maintains a limited-size cache that contains addresses of most recently bootstrapped peers. Every time a peer is bootstrapped, the address of this peer is automatically added to the cache, and peers are removed from the cache in FIFO order. This way the cache does not become stale. Peers that contact the bootstrap node for bootstrapping receive a random sample of peer addresses stored in the cache.

Figure 3.10 shows the bootstrap process of Peer A. First (1), Peer A obtains addresses of initial neighbours from the local cache. If any of the initial neighbours is alive, the peer skips the latter steps. Otherwise (2), the peer resolves well-known (hard-coded or manually set up) DNS names and obtains addresses of bootstrap nodes. From the bootstrap nodes (3), Peer A receives a list of addresses of initial neighbours. Finally (4), Peer A attempts to contact an initial neighbour (Peer B) and to obtain all information needed to join the system (neighbourhood sets, aggregates, etc.).

Optionally, every bootstrap node may run the same protocols as all other peers in the system. In this case, the bootstrap nodes can provide bootstrapped peers with complete initialisation data, such as aggregates, neighbourhood sets, etc.

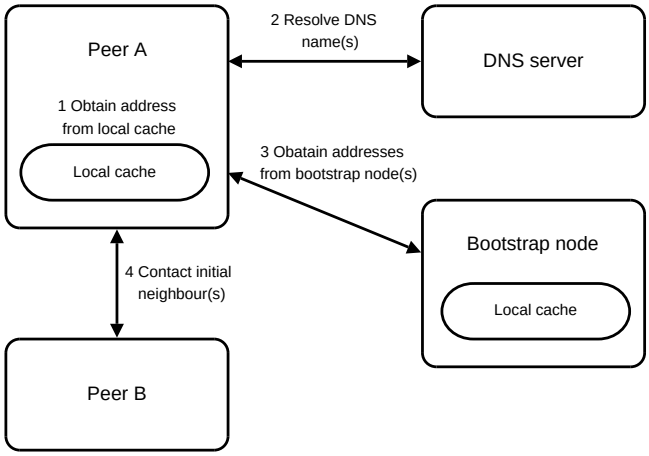

Figure 3.10: Bootstrap process of Peer A.

Peers may also use bootstrap nodes to detect whether they have open (unrestricted) IP addresses. In order to do so, a peer sends a request to selected bootstrap nodes and each bootstrap node attempt to connect to the requesting peer. If a connection is successfully established, the peer assumes that its IP address is not fire-walled. Otherwise, after waiting for a predefined amount of time, the peer assumes that its Internet access is restricted.

Chapter 4

Evaluation

4.1 Gradient Topology Evaluation

In this section, we describe the evaluation of the gradient topology. We describe the evaluation methodology, the experimental setup, and we present the results.

4.1.1 Evaluation methodology

We evaluate the algorithms used in the gradient topology by performing a number of experiments, in which we measure the performance of gradient search and the precision of the aggregation algorithm.

The following notation and metrics (based on the notation introduced in Section 3.1 describing the implementation of the gradient topology) are used in the remainder of this section. We measure the average error in histogram estimation, Err_H, defined as

$$Err_H(T) = \frac{\sum_{t=1}^{T} \sum_{p=1}^{N_t^*} D(H_t^*, H_{p,t})}{T \cdot N_t^*} \tag{4.1}$$

where N_t^*, $N_{p,t}$, Max_t^*, $Max_{p,t}$, $H_t^*$and $H_{p,t}$ correspond to N^*, N_p, Max^*, Max_p, $H^*$and H_p at time t of the experiment, T is the duration of the experiment, and D is a histogram distance function defined as

$$D(H_t^*, H_{p,t}) = \frac{\sum_{i=1}^{B} |H_t^*(i) - H_{p,t}(i)|}{\sum_{i=1}^{B} H_t^*(i)} \tag{4.2}$$

Similarly, we define Err_N as the average error in the estimation of N_t^*, and Err_{Max} as the average error in the estimation of Max_t^* over the course of the experiment.

We compare the performance of gradient search with random walking [33, 15] and with a probabilistic search strategy where a peer, x, selects the next-hop destination for a message with probability, P_x, given by the Boltzmann exploration formula [30]:

$$P_x(a) = \frac{e^{(U(a)/T)}}{\sum_{i \in N_x} e^{(U(i)/T)}}$$

where $P_x(a)$ is the probability that x selects neighbour a, $U(a)$ is the estimated utility of peer a, N_x is the set of x's available neighbours, and T is a parameter of the algorithm called the temperature. Setting T close to zero causes the algorithm to

be more greedy and deterministic, as in gradient search, while if T grows to infinity, all neighbours are selected with equal probabilities similar to random walking. Thus, the temperature enables a trade-off between exploitative (and deterministic) routing of messages towards the core, and random exploration that enables searches to escape local maxima.

Routing messages steeply towards the core, as in the gradient search, or Boltzmann search with a low temperature value, has the advantage over random walking that subsequent peers on a message's path are more and more stable, and therefore, the probability of message loss decreases.

For each of the searching algorithms we measure two properties. We calculate the average number of hops in which the algorithms deliver a search message from a random peer in the network to a super-peer in the core, i.e., to a peer above a certain utility threshold, and we measure the search failure rate, i.e., the percentage of search messages that are never delivered to super-peers.

The super-peer utility threshold is determined by each peer individually using the utility histogram calculated by the aggregation algorithm. A peer, p, sets the threshold, t_p, to a value that corresponds to 1% of highest utility peers. This value is approximated using the following formula

$$t_p = w \cdot \max_{1 \leq i \leq B} (H_p(i) \geq 0.01 \cdot N_p) \tag{4.3}$$

where w is the histogram width and B is the number of bins in the histogram.

4.1.2 Experimental setup

This subsection describes the experiments that we performed in order to evaluate the algorithms used in the gradient topology.

We ran our experiments on a Pentium 4 machine with a 3GHz processor and 3GB RAM under Debian Linux. We evaluate the aggregation and search algorithms in a Java-based discrete event simulator. An individual experiment consists of a set of peers, connections between peers, and messages passed between peers. We assume all peers are mutually reachable, i.e., any pair of peers can establish a connection. We also assume that it takes exactly one time step to pass a message between a pair of connected peers. We do not model network congestion, however, we limit the maximum number of concurrent connections per peer. In order to reflect network heterogeneity, we limit the number of peer connections according to the Pareto distribution (power law) with an exponent of 1.5 and a mean of 24 connections per peer.

The simulated P2P network is under constant churn. Every new peer p is assigned a session duration, s_p, according to the Pareto distribution with an exponent of $\gamma = 1.5$ and minimum value s_{min}

$$P(s_p > s) = \left(\frac{s_m}{s}\right)^{\gamma}$$

Thus, the expected session duration, $E(s_p)$, is given by formula

$$E(s_p) = \frac{\gamma \, s_{min}}{\gamma - 1}$$

We calculate the churn rate as the fraction of peers that leave (or join) the system at one step of the simulation. Over the lifetime of a running system, the average churn

rate, $E(c)$, is equal to the inverse of the expected peer session time $E(s_p)$, therefore, in order to simulate a churn rate, c, in the system, we set s_{min} to

$$s_{min} = \frac{\gamma - 1}{\gamma \cdot c} \tag{4.4}$$

We assume that 10% of peers leave the system without performing the leave procedure of the aggregation algorithm (i.e., they crash).

A bootstrap server is used that stores the addresses of peers that have most recently joined the network. The list includes "dangling references" to peers that may have already left the system. Every joining peer receives an initial random set of 20 neighbours from the bootstrap server. A peer discovers subsequent neighbours by performing the neighbour selection algorithm at every step of the simulation. If a peer becomes isolated from the network (i.e., has no neighbours), it is bootstrapped again. The bootstrap server executes the aggregation algorithm and provides initial estimates of N^*, Max^*, and H^*, for peers entering the system.

We start each individual experiment from a network consisting of a single peer. The number of peers is increased by one percent at each time step, until the network grows to the size required by the experiment. Afterwards, the network is still under continuous churn, however, the rate of arrivals is equal to the rate of departures and the number of peers in the system remains constant. Each peer continuously performs the neighbour selection and aggregation algorithms at every time step after it is bootstrapped. Additionally, at each turn, a number of randomly selected peers emit search messages that are routed using gradient search or random walking.

All peers attempt to either deliver search messages they hold in their buffers, if their utility is higher than the specified threshold, or forward messages to neighbours. If a peer's utility is higher than a defined utility threshold, all messages in its buffer are delivered. Otherwise, each message is forwarded to one of the peer's neighbours selected by the current search policy. When the TTL value of a message drops to zero, the message is discarded. Furthermore, if a peer leaves the system, all messages that it currently stores in its buffer are lost.

We examine peer churn rates between 0 and 0.1, where the value of 0.1 corresponds to a configuration where 10% of all peers leave the system at every step of the simulation. In physical time, if a discrete time step were 10 seconds, such a churn rate would correspond to roughly 1000 peer departures per second for a 100,000 peer network. We have observed that for extreme churn rates, such as 0.1 and higher, where the average peer session duration is very short, peers do not have enough time during their sessions to refine their initial neighbourhoods that they receive from bootstrap nodes, and in result, the network topology depends more on the bootstrapping method than on the neighbour selection algorithm run by peers.

For the purpose of the simulation, in all experiments, the number of bins in the utility histogram is 100 , the aggregation frequency parameter F is 10 (except Figure 4.1), and TTL is set to $3 \cdot log(N) + 10$ hops. The utility function of a peer p with uptime u and d maximum connections with neighbours is defined as $U(p) = d \cdot log(u + 1)$.

4.1.3 Evaluation Results

The experimental results show that the aggregation algorithm offers a good approximation of the system properties and that gradient search exhibits significantly better performance than Boltzmann searching and random walking, in terms of both number of hops and message loss rate.

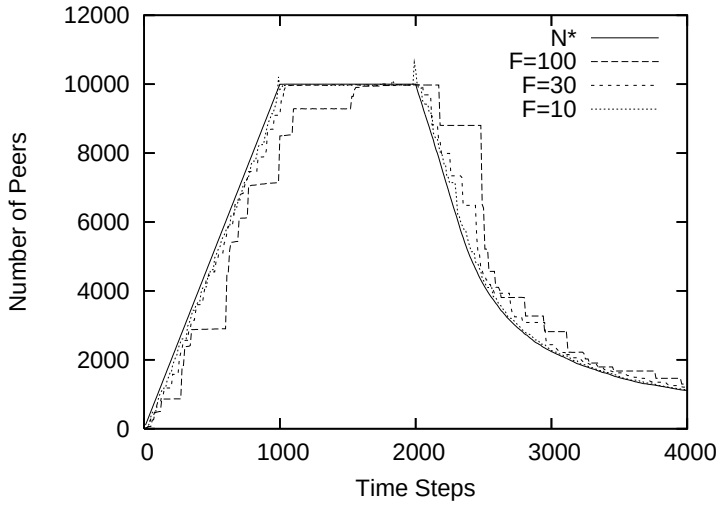

Figure 4.1: Average estimation of the number of peers in the system (N) as a function of time. Three experiments are compared, with the frequency of aggregation (F) set to 100, 30, and 10 time steps.

Figure 4.1 shows the average precision of N^* estimation as a function of time and compares the results obtained for three different values of F. The best approximation, close to N^*, is obtained for $F = 10$. Random fluctuations are visible.

Figure 4.2 and Figure 4.3 show the average error of the aggregation algorithm, Err_N, Err_{Max}, and Err_H, as a function of the churn rate and as a function of the network size. The variance is not shown as it is approximately two orders of magnitude lower than the plotted values. The churn rate is measured as the number of substituted peers per time step. The estimation of Max^* is the most precise as the algorithm for the maximum calculation is simpler compared to the algorithm for H^* and N^* estimation. H^* approximation is less accurate than N^* since the histogram changes more dynamically than the number of peers. The relative error as a function of the number of peers is bounded as the number of rounds in the epoch is proportional to $log(N)$, which corresponds to the theoretical analysis of Kempe [13].

Figure 4.4 shows the average hop count for delivered messages as a function of the network size. The churn rate was fixed at 0.01. We can see that gradient search performs better than other search strategies, and that the message hop count increases together with the Boltzmann temperature. For the random walk, the hop count grows more slowly than for gradient search with increasing network size. This can be explained by the fact that the average number of high utility peers is a fixed percentage of the network size, and hence, the probability of high utility peer discovery by random walking is a function of this percentage. For gradient search, the hop count increases with the network size, since the average distance from a peer to the core increases. We

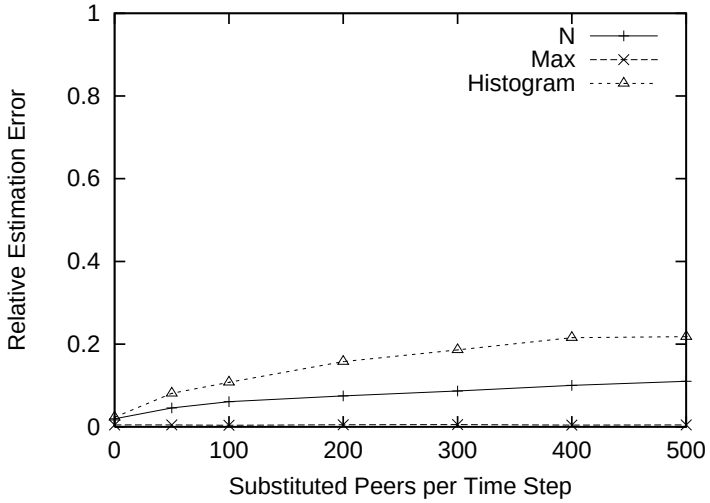

Figure 4.2: Average estimation error of the number of peers in the system (N), maximum utility (Max), and the utility distribution (Histogram) as a function of peer churn rate.

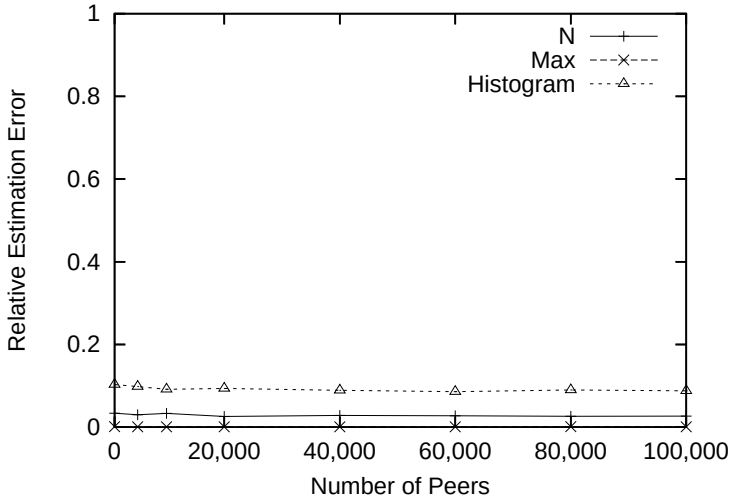

Figure 4.3: Average estimation error of the number of peers in the system (N), maximum utility (Max), and the utility distribution (Histogram) as a function of network size.

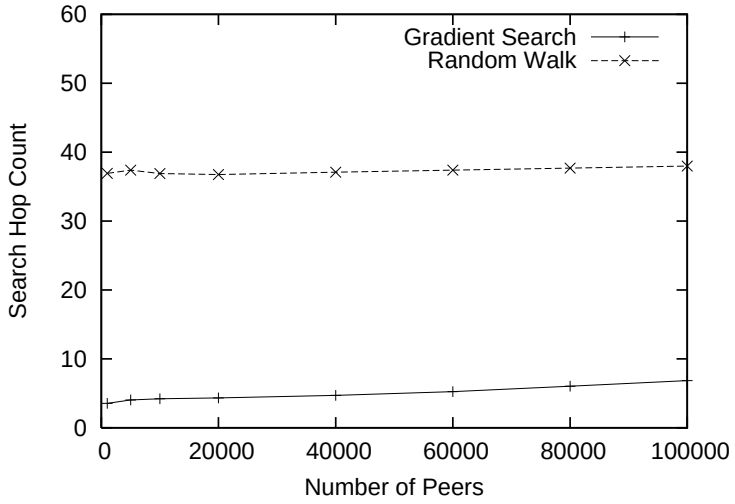

Figure 4.4: Average hop count of delivered messages as function of network size with a churn rate of 0.01 and TTL set to 100.

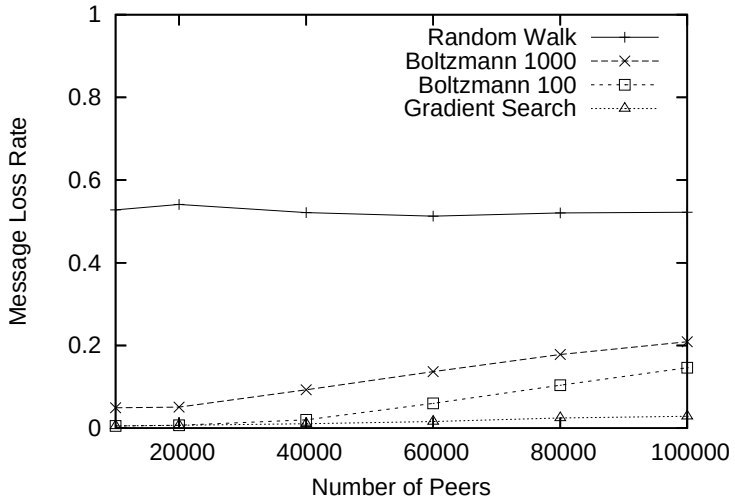

Figure 4.5: Average message loss rate as function of network size with a churn rate of 0.01 and TTL set to 100.

can also see that the two Boltzmann approaches, with different temperatures, converge as the network size grows. This is due to the growing average utility (uptime) of peers in the system, which results in a decreasing relative difference between the Boltzmann temperatures.

Figure 4.5 shows the average message loss rate as a function of the network size with a churn rate of 0.01, and Figure 4.6 shows the average message loss rate as a function of the churn rate for a network of 10,000 peers. Both figures demonstrate that the message loss rate is lowest for the gradient search, and that it grows as the Boltzmann temperature is increased.

Better performance of the gradient search results from two facts. First, as shown in Figure 4.4, the message path is shorter in gradient searching than in other search strategies, and therefore, the probability that a message is lost by forwarding peers, or that the message exceeds its TTL value, is lower. Second, as confirmed by measurements reported below, the stability of peers used for forwarding messages in the gradient search is higher, which additionally reduces the message loss probability. For random walking the message loss rate is nearly equal for all network sizes, which is due to the fixed percentage of high utility peers in the system.

Figure 4.7 presents the message loss rate as a function of the churn rate with a distinction between message loss caused by exceeded message TTL and message loss caused by peers leaving the system. The total message loss rate is calculated as a sum of the two mentioned loss rates. The figure shows that for random walking the message loss rate attributed to peers leaving the system grows together with the churn rate. At the same time, the message loss rate attributed to exceeded TTL decreases with growing churn, which means that for higher churn rates messages are more likely to be lost by leaving peers than by exceeding their TTL values. For gradient search, nearly 100% of the total message loss is caused by churn, i.e., messages hardly ever reach their maximum TTL value.

Figure 4.8 shows the message loss rate as a function of message TTL. We can see that the overall message loss decreases when TTL grows. For random walking, as the TTL is increased, messages are lost more often due to churn, i.e., because of peers leaving the system. As a consequence, the message loss rate does not converge to zero. On the contrary, for the gradient search, the message loss rate becomes negligible for TTL values above approximately 50 hops.

Figure 4.9, Figure 4.10, and Figure 4.11 demonstrate the average utility of peers used for forwarding messages in different searching strategies. In all cases we can see that the average hop utility is highest for gradient search and lowest for random walks. This result is consistent with the observation that for gradient search the message loss rate is lower than for the other strategies. In Figure 4.10 and Figure 4.11 the utility is scaled in such a way that the average utility over all peers in the system is 1. As expected, for random walking the average path utility is 1. Figure 4.9 shows also that the average peer utility (measured as uptime) grows steeply when the churn rate approaches zero.

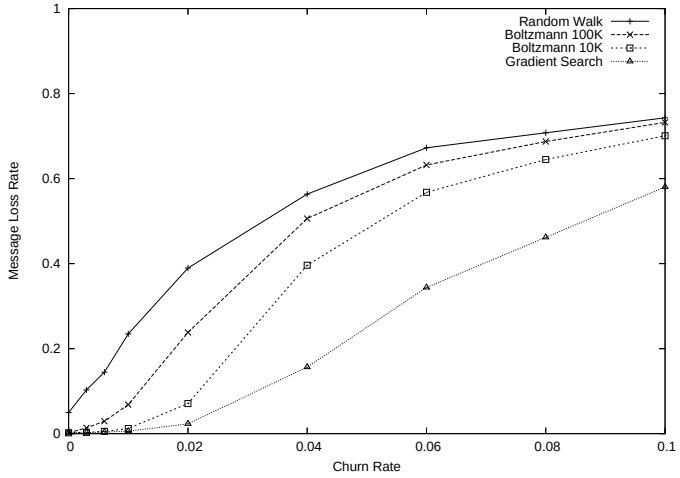

Figure 4.6: Message loss rate as a function of peer churn rate. Comparison between random walk, Boltzmann search, and gradient search (network size is 10,000 and TTL=100).

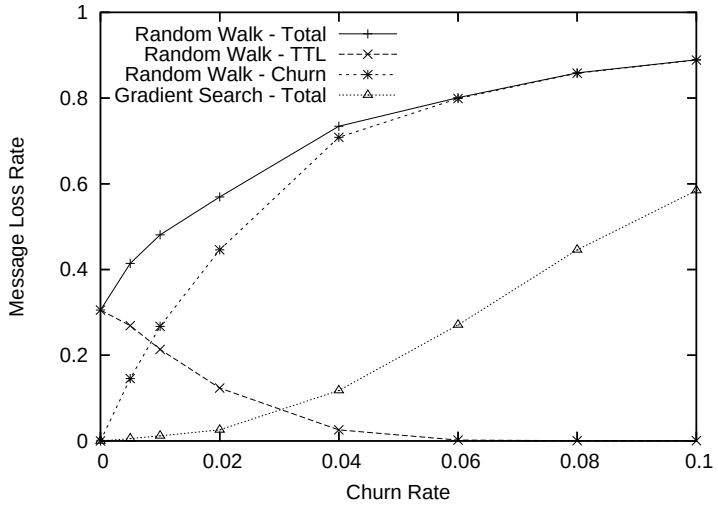

Figure 4.7: Message loss rate as a function of peer churn rate. Comparison between message loss rates attributed to exceeded message TTL and message loss attributed to peer churn. For gradient search, nearly 100% of the observed message loss is caused by peer churn (network size is 10,000 and TTL=100).

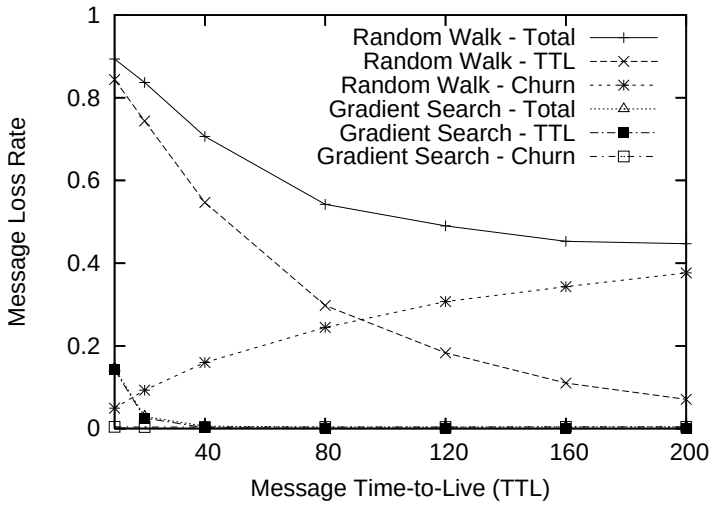

Figure 4.8: Message loss rate as a function of message TTL for 10,000 peers and 0.01 churn rate. The graph shows a distinction between message loss caused by exceeded message TTL and message loss caused by peers leaving the system (network size is 10,000, TTL=100).

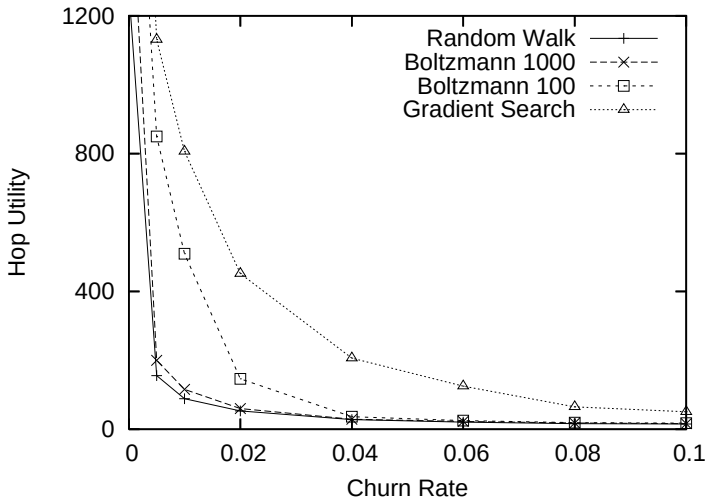

Figure 4.9: Average utility of peers forwarding messages (hop utility). The average utility of all peers in the system, measured as uptime, decreases with the churn rate. Gradient search achieves better hop utility by forwarding messages to the highest utility peers (network size is 10,000, TTL=100).

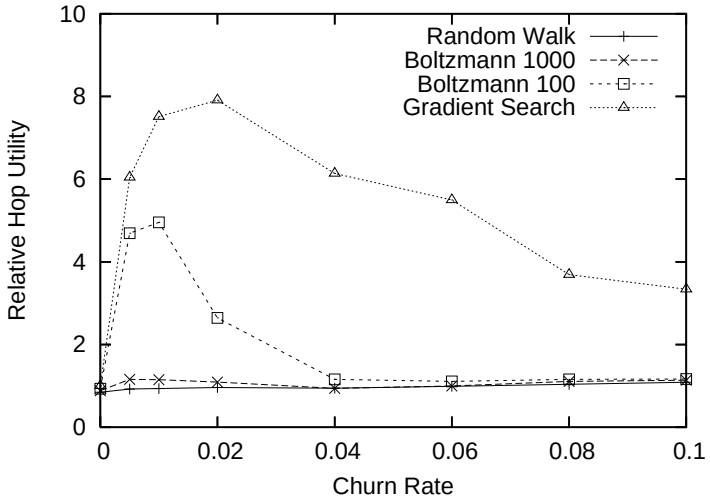

Figure 4.10: Average relative utility of peers forwarding messages (relative hop utility) as a function of churn rate. The utility is scaled so that the value of 1 corresponds to the average utility among all peers in the system.

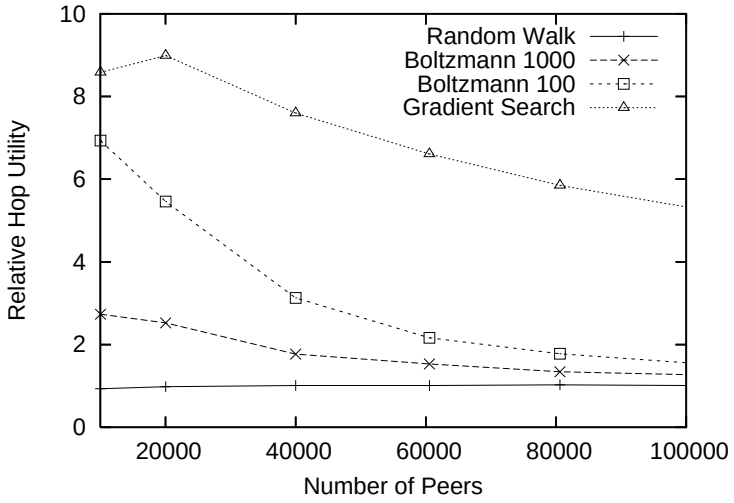

Figure 4.11: Average relative utility of peers forwarding messages (relative hop utility) as a function of network size. The utility is scaled so that the value of 1 corresponds to the average utility among all peers in the system.

4.2 Distributed Hash Table Evaluation

Distributed Hash Tables and in particular the Bamboo DHT protocol have been extensively evaluated in previous studies [23, 24, 22, 29, 25, 21, 16]. DHTs have been shown in both theoretical and experimental evaluations to exhibit a number of nice properties such as routing in $O(logN)$ hops (where *N* is the number of nodes in the system), good resilience to peer churn and adaptability to changing network conditions. Therefore, our work has focused evaluating our DHT extensions that enable users to store and retrieve larger data values and improve the security properties (see Section 3.2). We have evaluated these extensions on the PlanetLab wide-area network testbed [2] that consists of about 700 computers spread all over the world and is available for researchers to investigate decentralised systems. A deployment of the Bamboo protocol is deployed on PlanetLab in the form of OpenDHT [24]. There are about 200 machines at any time available in the OpenDHT overlay network.

In the experiments we used a machine in the Trinity College network (**.cs.tcd.ie*) that was acting as a *client process* (see the DHT architecture in Figure 3.8) and was issuing *put* and *get* operations to respectively *store* and *retrieve* proxy objects in the DHT overlay. The gateway (*server process*) has been selected to be in Intel, Cambridge, UK (*planetlab1.cambridge.intel-research.net*). The proxy objects were 1kB, 10kB, 100kB and 200kB in size, each experiment has been repeated 15 times and an average over all trials has been calculated. The results are presented in Figure 4.12.

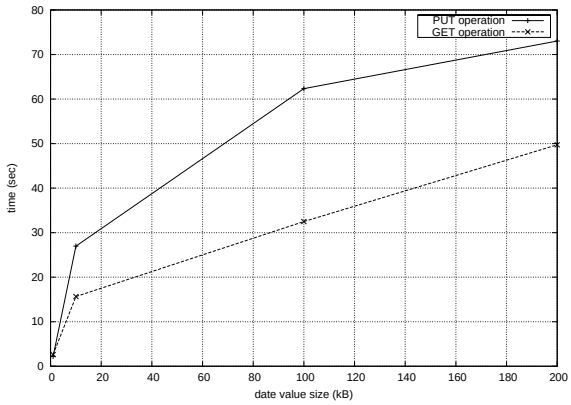

Figure 4.12: Results of put and get operations on the DHT overlay

It can be seen from the figure that get operations had better performance than put operations. The put operations have been the first to be invoked. For this reason, nodes in the overlay have been able to learn the best routing paths to nodes that stored the individual chunks of the data values. This knowledge substantially improved the performance of subsequent get requests for the same chunks. The results also show a nice feature of the DHT overlay, where the latency of operations grows slower than size of data values. This is a result of a parallelisation of our DHT extensions for large file routing and the fact that much overhead is associated with establishing the RPC connections.

Chapter 5

Conclusion

This document introduced a dual-topology for Service-Oriented Architectures to support a self-organising service broker placement strategy and a decentralised approach to store service endpoint information. The architecture is an extension to the servent application server and is encapsulated as a core-service. It exposes an abstract interface in order to deploy, discover, and undeploy services and to be able to join and leave the P2P network. The design of the architecture allows for future improvements and extensions without breaking backwards compatibility by hiding the core functionality behind interfaces and abstract classes.

Also an extensive evaluation was provided and mainly focusing on the Gradient Topology, because it introduced novel concepts that have not been implemented before for SOAs, while utilising the DHT takes advantage of an existing and stable body of work that has been adopted from both, academic and industrial projects. In particular, this document clarifies many issues that were raised during the review in Tampere January 2006.

Gradient Topology We have proposed a gradient network topology where the highest utility peers are highly connected with each other and form a logical core suitable for maintaining a system's critical infrastructural services. A self-organising neighbourhood selection algorithm has been presented that generates the gradient topology by clustering peers with similar utility characteristics. The main advantage of the gradient topology is that the network structure contains information about the peer utility, which allows the peers to discover other high utility peers without flooding the entire network with search messages. The gradient topology allows the search space to be limited to a small subset of all peers in the system. Furthermore, decentralised aggregation techniques are used to reduce the uncertainty about the system by approximating peer utility distribution, and enable the decentralised calculation of adaptive utility thresholds. We have shown that the gradient topology in combination with a peer utility metric, aggregation techniques, and gradient searching allows efficient discovery of high utility peers in open, decentralised and heterogenuous peer-to-peer systems.

The topology can be used to improve the availability and performance of system's infrastructural services by placing them on the highest utility peers, as well as to reduce the amount of network traffic required to discover and use these services. The topology enables a trade-off between centralisation and decentralisation, in the sense that it allows the selection of a subset of high utility peers for supporting application services rather than distributing the services equally between all peers.

The evaluation of our work shows that gradient search achieves significantly better performance than random walking. Our results agree with the no-free lunch theorem for search [32] that states that no generalised search algorithm, such as random walking, can out-perform a specific search algorithm that makes use of suitable domain knowledge. The gradient topology contains implicit knowledge of peers' utilities and this knowledge is exploited by our gradient search algorithm, enabling its significant performance gains over random walking. Moreover, measurement results show that the neighbour selection algorithm is scalable and robust, and that it generates a topology with a very low diameter. Our experiments also confirm that the aggregation algorithm allows a precise approximation of system-wide peer properties.

Distributed Hash Tables We have shown how the Distributed Hash Table overlay can be used for service proxy storage, location and removal. We explained how the properties of DHTs enable to achieve these tasks in a decentralised peer-to-peer environment. We have identified limitations of the existing DHT implementations that we addressed in order to adopt them to the DBE requirements. Our work on these limitations resulted in a DHT extension layer that allows for: a private namespace for each applications using the DHT in order to avoid conflicts of keys between different applications; storing and retrieving data values of arbitrary sizes; protecting all vulnerable user data with a single user private key.

Bibliography

[1] *Service-oriented computing: concepts, characteristics and directions*, 2003. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1254461`.

[2] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, 2003. ISSN 0146-4833.

[3] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the 1st International Workshop on Designing Privacy Enhancing Technologies*, pages 46–66, 2000.

[4] Bram Cohen. Incentives Build Robustness in BitTorrent. In *Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems*, 2003.

[5] Eoin Curran and Jim Dowling. SAMPLE: Statistical network link modelling in an on-demand probabilistic routing protocol for ad hoc networks. In *Proceedings of the 2nd Conference on Wireless On demand Network Systems and Services*, pages 200–205. IEEE Computer Society, January 2005.

[6] G. Di Caro, F. Ducatelle, and L.M Gambardella. AntHocNet: An adaptive nature-inspired algorithm for routing in mobile ad hoc networks. *European Transactions on Telecommunications, Special Issue on Self-Organization in Mobile Networking*, 16:443–455, 2005.

[7] C Diot, B Neil Levine, B Lyles, H Kassem, and D Balensiefen. Deployment issues for the IP multicast service and architecture. *IEEE Network*, 14(1):78–88, 2000.

[8] Per Brinch Hansen. *Operating System Principles*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1973.

[9] Charles Antony Richard Hoare. Monitors: An operating system structuring concept. *Communications*, 17(10):549–557, Oct 1974.

[10] Michael N. Huhns and Munindar P. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, January 2005. ISSN 1089-7801. URL `http://dx.doi.org/10.1109/MIC.2005.21`.

[11] M. Jelasity and O. Babaoglu. T-man: Gossip-based overlay topology management. In *the 3rd International Workshop on Engineering Self-Organising Applications*, 2005.

[12] Márk Jelasity and Alberto Montresor. Epidemic-style proactive aggregation in large overlay networks. In *Proceedings of the 24th International Conference on Distributed Computing Systems*, pages 102–109. IEEE Computer Society, 2004.

[13] David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *Proceedings of the 44th IEEE Symposium on Foundations of Computer Science*, pages 482–491, October 2003.

[14] J Kubiatowicz, D Bindel, Y Chen, S Czerwinski, P Eaton, D Geels, R Gummadi, S Rhea, H Weatherspoon, W Weimer, C Wells, and B Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the 9th international Conference on Architectural Support for Programming Languages and Operating Systems*, pages 190–201, 2000.

[15] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th International Conference on Supercomputing*, pages 84–95, 2002.

[16] Gurmeet Singh Manku, Mayank Bawa, and Prabhakar Raghavan. Symphony: Distributed hashing in a small world. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, pages 127–140, 2003.

[17] Alper Tugay Mizrak, Yuchung Cheng, Vineet Kumar, and Stefan Savage. Structured superpeers: Leveraging heterogeneity to provide constant-time lookup. In *Proceedings of the 3rd IEEE Workshop on Internet Applications*, pages 104–111, 2003.

[18] Alberto Montresor. A robust protocol for building superpeer overlay topologies. In *Proceedings of the 4th International Conference on Peer-to-Peer Computing*, pages 202–209, 2004.

[19] Alberto Montresor, Márk Jelasity, and Ozalp Babaoglu. Robust aggregation protocols for large-scale overlay networks. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 19–28, June 2004.

[20] Ruggero Morselli, Bobby Bhattacharjee, Aravind Srinivasan, and Michael A. Marsh. Efficient lookup on unstructured topologies. In *Proceedings of 24th ACM Symposium on Principles of Distributed Computing*, pages 77–86, 2005.

[21] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. Load balancing in structured p2p systems. In *the 2nd International Workshop on Peer-to-Peer Systems*, 2003.

[22] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *Proceedings of the Conference on Applications, Technologies, Trchitectures, and Protocols for Computer Communications*, pages 161–172, 2001. ISBN 1-58113-411-8.

[23] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a dht. In *Proceedings of the USENIX 2004 Annual Technical Conference*, pages 127–140, 2004.

[24] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. Opendht: a public dht service and its uses. *SIGCOMM Comput. Commun. Rev.*, 35(4):73–84, 2005. ISSN 0146-4833.

[25] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th International Conference on Distributed Systems Platforms*, pages 329–350, 2001.

[26] Jan Sacha and Jim Dowling. A self-organising topology for master-slave replication in p2p environments. In *Proceedings of the 3rd International Workshop on Databases, Information Systems and Peer-to-Peer Computing*, pages 52–64, July 2005.

[27] Jan Sacha, Jim Dowling, Raymond Cunningham, and René Meier. Discovery of stable peers in a self-organising peer-to-peer gradient topology. In *Proceedings of the 6th IFIP International Conference on Distributed Applications and Interoperable Systems*, number 4025 in LNCS, pages 70–83. Springer-Verlag, June 2006.

[28] Jan Sacha, Jim Dowling, Raymond Cunningham, and René Meier. Using aggregation for adaptive super-peer discovery on the gradient topology. In *Proceedings of the 2nd IEEE International Workshop on Self-Managed Networks, Systems & Services (SelfMan)*, number 3996 in LNCS, pages 77–90. Springer-Verlag, June 2006.

[29] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Computer Communication Review*, 31(4):149–160, 2001. ISSN 0146-4833.

[30] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[31] D. Tsoumakos and N. Roussopoulos. A comparison of peer-to-peer search methods. In *Proceedings of the 6th International Workshop on the Web and Databases*, 2003.

[32] David H. Wolpert and William G. Macready. No free lunch theorems for search. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[33] Beverly Yang and Hector Garcia-Molina. Improving search in peer-to-peer networks. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 5–14, 2002.

[34] Beverly Yang and Hector Garcia-Molina. Designing a super-peer network. In *Proceedings of the 19th International Conference on Data Engineering*, pages 49–60, 2003.

[35] Ben Y. Zhao, Yitao Duan, Ling Huang, Anthony D. Joseph, and John D. Kubiatowicz. Brocade: Landmark routing on overlay networks. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, pages 34–44, 2002.