Design and Evaluation of an Anonymization Service for Structured P2P Overlays

Bachelor Thesis

by

Andrej Morlang

born in
Karaganda

submitted to

Technology of Social Networks Lab
Jun.-Prof. Dr.-Ing. Kalman Graffi
Heinrich-Heine-Universität Düsseldorf

Oktober 2014

Supervisor:
Jun.-Prof. Dr.-Ing. Kalman Graffi

Abstract

This thesis describes the anonymization services for the Chord-Overlay.

The original *Chord* [10] design is focused on fast performance of routing, which means that each request can be found in $O(logN)$ steps. This Chord design does not consider the anonymity aspect for the given routing model. Therefore, existing anonymity routing techniques has been reviewed. According to this knowledge, different designs are created, but because of the depth impact of the necessary changes in the implementation, the already researched solution *AChord* [12] is regarded firstly. While AChord contains disadvantages on the join process and performance, AChord approach is modified into the "SChord" in this thesis. The new anonymous service SChord is implemented in the given simulator *PeerfactSim* [8] as well as the original AChord service. Further, the simulator already contains the Chord overlay. The overlay design has three weak point where the anonymity is threatened. Those are the joining process, the *finger table* [10] update and the routing process. The joining process and the finger table are specified by the overlay, while the routing process may be adapted in some ways. The routing itself contains further three anonymity aspects, sender anonymity, responder anonymity and the intercommunication anonymity between the sender and responder.

To test these anonymous conditions each service is modified by a spy operation. The spy operation can act passively or/and actively, to differentiate between the types of attack and to achieve proper evaluation results. Each service Chord, AChord and SChord is tested by active and passive type of eavesdropper, and the results of the spy process are compared on these three services.

The given results represents that SChord is the most anonymous service while it knows the minimal number of participants on the overlay in all anonymity aspects, in comparison to the AChord and Chord services. The AChord service provide more anonymity on the routing process than Chord but because of the disadvantages in the design of the join process it may collect, under certain circumstances, more information of the network than Chord. Still AChord provide anonymity services while Chord does not concern about the anonymity at all.

Acknowledgments

I would like to thank my family and friends for their support. Especially I would like to express my gratitude to my advisor Tobias Amft for the correcting the grammar and style, and advising me on this bachelor thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Decentralized peer-to-peer overlays are often used as file sharing network. Compared to centralized networks, decentralized networks are almost invulnerable against breakdowns, since the information is stored distributed on different nodes of network. Each member expands the capacity of the existing network by providing a part of its own disk space to it, so the network remains scalable. To achieve efficient search of the content, different approaches have been developed e.g. *Pastry, Chord, Kademlia*. Still the mentioned overlays are not concern about the anonymity of participants inside the overlay. In general, anonymity means that each user is allowed act on the network, so that the identity of the acting user remains unknown to the whole network. This anonymity aspect can be divided further by three states: sender anonymity, receiver anonymity and the anonymous intercommunications between sender and receiver. Sender anonymity is when the sender of a specific content can not be (clearly) defined by any member of the network. The receiver anonymity is when the holder of specific content can not be (clearly) defined by any member of the network. The anonymous intercommunications between sender and receiver means that the initiator must remain unknown to the responder as well as the responder hast to be unknown to the initiator. Those three anonymity aspects may appear to be identical but they are not, since each aspect has its own technique to achieve the required anonymity condition. The two implemented approaches from Chapter 4, try to fulfill these anonymity condition. Additionally an eavesdropper (spy-node) exist to test the implemented approaches on those anonymity aspects. The loss i.e. nonexistence of anonymity may tracing further consequences to the data stored on the network. If the source (e.g. responsible node) can be clearly defined, it is possible to remove the data form the network by replacing the sources storage. Because of the design, members must store particular information even if they do not request it. A member is always responsible for the information it stores. Basically, any corrupted content may be stored on any members node, if this node is known to the attacker. This opens a new facet why the anonymity in these overlays must be included.

There are few anonymity approaches existing for peer-to-peer overlays in general, and even less for structured peer-to-peer overlays. Mainly because the anonymization procedure is in contrast with the efficient search for the given overlay. While the members in the overlay remain anonymous to each

other, the efficient routing is difficult to accomplish.

Based on the given simulator *PeerfactSim* [8], "SChord" anonymity service for a structured peer-to-peer overlay is developed in this bachelor thesis and evaluated with the *AChord* [12], an already existing solution. The remainder of this thesis is structured as follows. Chapter 2 represents related work, which describes existing anonymity methods and point out the differences, disadvantages and advantages of them. Chapter 3 represented several anonymity designs. "Friend in the Middle", "Single User", "Friends" and "SChord" are developed up by me, according to the given information from Chapter 2 and from the *AChord* approach. *AChord* is an already existing solution, represented in [12], which I did not developed. Chapter 4 describes the simulator structure and the implemented anonymity services *AChord* and "SChord", as well as the testing conditions. Where Chapter 5 evaluate those states. In Chapter 6 the results are summed up.

Chapter 2

Related Work

A short insight and integration of existing anonymity routing methodes is give in this Capter.

Existing anonymity routing methods can be classified by three categories.

- “Trustful Environment” - services relying on the aspect of the community.
- “Discreet User” - services that share no information about the user.
- “Indirect Swarming Communication”- services which are flooding the network until the responder replies.

Crowds [1] relies on trustful environment since a user communicates via other participants. The member remains invisible for the end target but every participant can read the messages forwarded to the destination and source. *Crowds* is categorized as the Trustful Environment service. Although decryption of the routed content i.e. message, does not directly provide anonymity, some application are supported by it. *Onion routing* [5] is one of them. In some cases it is similar to *Crowd's* routing method by the random path transmission with two main differences: Before forwarding the message, a proxy strips away any user specific information and when forwarding a message, it is always protected with a key of the next user on the path. Another discreet user service is *Mixes* [3], which equally to *Onion routing* encrypts the messages on the routing path. Further, *Mixes* mix the users among themselves to provide more anonymity. *Mixes* can be modified to provide the receiver anonymity as well, but this approach is usually not used for the real time communication. *Onioin routing* and *Mixes* are categorized in the Discreet User service. The represented methods provide only the anonymity for the initiator of the request but not for the responder. This is others in the indirect swarming communication services, where the receiver remains unknown as well as the initiator. There both sides communicate indirectly with each other. The message is e.g. forwarded via a key to the next participant closer to this key, but it remains unknown who is the real initiator and responder. Those methods

are represented in *Ants* [20] [2] and on *Freenet* [7]. *Freenet* and *Ants* are categorized as the Indirect Swarming Communication service. In the following, these routing methods will be explained.

2.1 Crowds

2.1.1 Working principle

Crowds working principle is to hide a user in a group of participants. The request travels along by random number of users with crowd membership. When a user finally decides to send the request to the *"End Server"* a path is established. End Server is defined as any website or webserver where a user sends the request to. The decision of sending the request depends on the forwarding probability, which itself is related to the number of Crowds participants. Since the origin remains unknown, because of the path, each user sending a request appears to the end server more like any other user than like a request initiator.

2.1.2 Anonymous-Routing principle

To join *Crowd* a user must registers himself on a server, called *blender* [1]. The blender maintains all members of the crowd, decides whether a user may, or may not join the crowd i.e. group and is responsible for the entry point and time, which is not related to regular time. The time described here is how long the blender needs to wait until the responses of the crowd members reaches the destination. When responses travel to long the blender is able to cut the path where the response should be routed through. Further, in this time new requests are disabled. When the blender accepts a user as a member of the crowd, it assigns a unique symmetrical key to this user. Further, it informs the crowd that this user is going to join in. By doing this, all members first reset the existing paths and then create new ones. This is how the newly joined members remains hidden. Now, the user who has the membership, initiates the request and sends it to a random member of the crowd. Each member has a list of active users wherefrom a random user is chosen. A user that receives the request has two options, forwarding the request to the end server or passing it to another randomly chosen active user(self-included), shown in Figure 2.1. If an active user does not respond, it is removed from the list. These steps are repeated as long as the request does not reach the End Server. The probability of forwarding a request to the End Server is related to the number of participants in *Crowds*. To avoid similar behavior pattern and loops, each user possesses an identifier (ID), a random 128-Bit value, which changes when predecessor ID remain the same. The response returns backward, by following the path it has been sent before.

2.1.3 Anonymity degree

According the research [1] of M. K. Reiter and A. D. Ruben Crowds provides the following anonymity properties shown in Table 2.1.
n is the number of crowd participants.
p_f is the probability for forwarding the request to the end server and must be in the range between $1/2 \leq X \leq 1$.
c is the number of collaborating members along the path.

Attacker	Sender anonymity	Receiver anonymity
local eavesdropper	exposed	P(beyond suspicion) $\lim_{n\to\infty} = 1$
c collaborating members $n \geq \frac{p_f}{p_f - 1/2}(c+1)$	probable innocence P(absolute privacy) $\lim_{n\to\infty} = 1$	P(absolute privacy) $\lim_{n\to\infty} = 1$
end server	beyond suspicion	N/A

Table 2.1: Crowd degree of anonymity [1]

The left column describes the sort of attack, the middle and the right column represents how much anonymity is offered against the specific attack. The likelihood degree of anonymization directly depends on size of the crowd and partly on the forwarding probability. Since these values vary for every crowd, a constant degree of anonymity can not be represented. In general it is said: the more members participate, the higher is the chance to remain unknown.

2.1.4 Anonymous-Routing example

For visualization reasons each initiator of the request and each webserver to whom a request is sent have the same number see Figure 2.1. Each number inside the circle is related to the same number inside the square, although in the algorithm the webserver is not aware of the initiator. Each arrow type represents different paths for every initiator and receiver. Each node knows only the predecessor and the successor (if any exists). When joined, the crowd members creates new paths. Depending on the forwarding probability, each member decides whether to route the request to web server or to another member. Considering, from Figure 2.1, the member with the *node-ID 3* as initiator. First, a random node where to forward the request to is chosen. In this case, it is the *node 1*. Note that *node 1* does not know if *node 3* is the initiator or another request transmitter. After receiving a request *node 1* decides to pass it further to *node 6*. This is where the request is finally send to the *web server 3*. For the web server, *node 6* appears as the requester and the real initiator remains unknown.

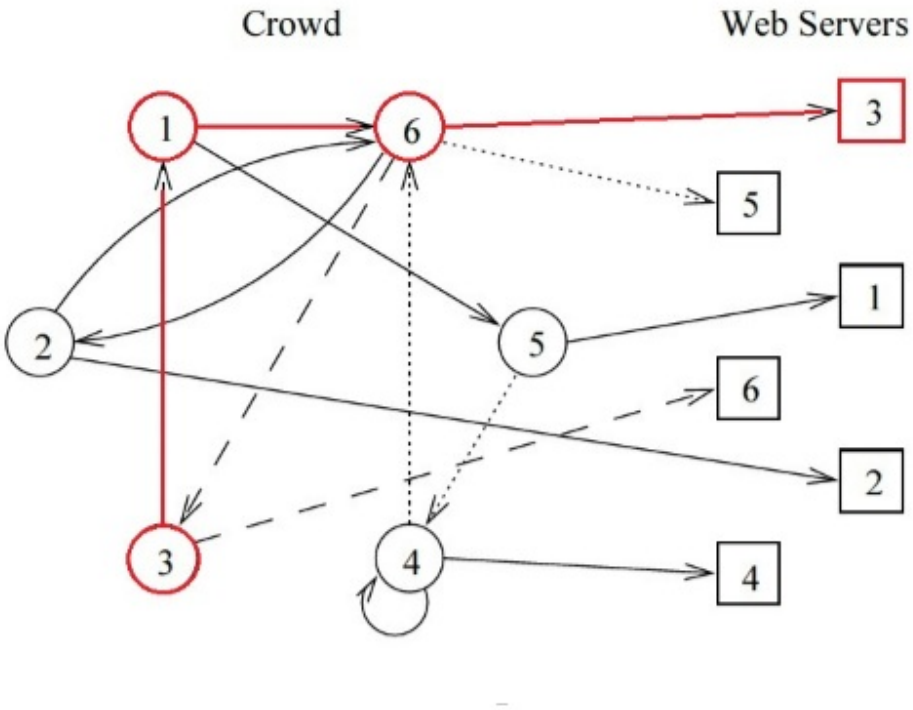


Figure 2.1: Routing example of *Crowds* [1]

2.2 Ants

2.2.1 Working principle

ARA [20] is a routing algorithm, developed for *MANETs*. *MANET* is a network, which provides mobile communication via radio and without any infrastructure knowledge. Since mobile nodes and radio have limited transmission range, it is necessary for each node to forward the content through other nodes. Based on the *ARA* algorithm [20], the *Ants* [2] application is developed. This *Ants* approach relies on *swarm intelligence* where subsets of swarms solve complex problems by cooperation [2]. The basic algorithm marks every node, by forwarding the request as well as the response, from source to destination on the path. The more often a node is visited the more marks it has. On the opposite, not used nodes decrease the amount of marks. Considering this, all nodes with the highest mark build a path. The most often marked path must be the shortest since the information traveling by several nodes is faster than by all the others.

2.2.2 Anonymous-Routing principle

Every marked node has a *pheromone* value, which is actually the mark itself [2]. Transferring information through the node increases the pheromone value by a static pre-defined number. Every node maintains a list of neighbors. When a route is established, the next hop is considered as the neighbor with the highest pheromone value. This value decreases when the destination is not available or a shorter path exists but also over period of time. The routing algorithm consists of three steps, which are *route discovery*, *route maintenance* and *route failure handling* [2].

Route discovery is responsible for creating a new path. Therefore, *FANTs* (forward ants) and *BANTs* (backward ants) are used [2]. The FANT is a small packet with a unique sequence number. Nodes, which receive a FANT, take a record in the routing table. The properties are read as following: *destination address* is the source address, *next hop* - the next node and *pheromone value* as a necessary number of hops for the FANT to reach the node [2]. The FANT is forwarded via the neighbors until it reaches the destination. The node destroys duplicated FANTs that contain the same sequence number and destination address (if a record exist). At the destination, a FANT is decrypted and destroyed. As a result, the BANT will be send to the source node. BANTs have the same routing principle as FANTs. When source node receive the BANT, the data transmission follows.

Route maintenance handles the path changes as well as improvements when indicated. These changes occurs through movement of mobile devices, which changes the length of the circuit as well as other properties. The pheromone value increases during the process of path building and by the amount of packages sent through a node. The value decreases, over time when the node routes no packets through, or by detecting a loop. Loops can occur when several paths leads to the same destination where the distance on the paths are none equal. As a result a node gets at least a double requst, which actually represents a loop. However, the method provides a sleep feature to prevent traffic overload. If the pheromone amount reaches a threshold, the node falls asleep. While sleeping, the sleeping node will not accept any pakets from other nodes, except those that are destined to the node itself.

Route failure handling recognizes route failures caused by missing acknowledgments. In this case, the pheromone value is set to zero. The node tries to send the data packet via an alternative route if a record exists, otherwise it informs the previous neighbor. Those precede on the same procedure. If a packet is backtracked to the source node, it imitates a new route discovery phase.

2.2.3 Anonymous-Routing example

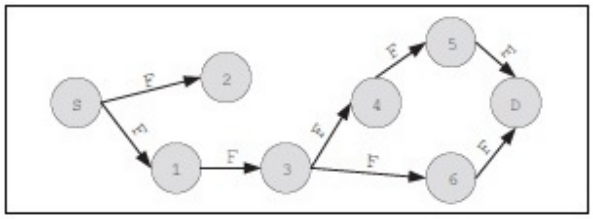


Figure 2.2: FANTs routing [2]

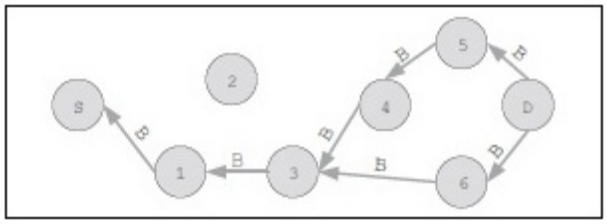


Figure 2.3: BANTs routing [2]

Figure 2.2 and 2.3 describe the process of establishing a route. The sender *S* initiates a request, since no route exists, *S* sends the request (FANT) to all its neighbors. *Node 2* does not contain the requested information nor has it any neighbors. Consequently, it sends the request back to predecessor and decreases the pheromone value to zero. *Node 1* does not contain the requested information as well but a neighbor is existing, so the request is forwarded to it. *Node 3* routes the request to *node 4* and *6*. If we assume that every connection has the same length, as a matter of fact the request from *node 6* already reaches the destination point when *node 5* gets the information from *node 4*. The duplicated request that follows from *node 5* to *node D* will be destroyed and the pheromone value of *node 5* decreases to zero. Following the behavior of *node 5*, *node 4* decreases the value to zero as well. Branches with zero pheromone concentrate are invalid. *Nodes 1, 3* and *6* increases the pheromone value. Now when *node D* decrypts the message it floods the network again toward to *node S*. Again, the values of *node 1, 3* and *5* increase and the path is finally established.

2.3 MIXes

2.3.1 Working principle

To provide anonymity, every message must pass at least one *Mix* or a *Cascade* [3]. A Cascade represent a row of Mixes to provide higher anonymity degree. Receiving a particular amount of requests, a Mix transforms the input of each request, so that the output is related to another initiator. Depending on network state and amount of users, messages can be sent continuously or periodically. Every

message gets encrypted and can be split into multiple items. Each initiator supplies the same amount of messages to each Mix to prevent analysis of network traffic. A set of specific number of Mixes or dummies requests, can be defined by a user, to reduce the traffic on the network, although it may cause negative effect on the anonymity.

2.3.2 Anonymous-Routing principle

The approach on Mixes is to transform the input and the output of the message. First, a pair of keys are created K^+ (public key) and K^- (private key). By encrypting the message $K^+(M)$ with a public key, only the holder of a private key is able to decrypt the content. Additionally, a random string of bits, here called R, is attached to the message. A is assigned as the destination address. The transformation by the Mix from input into the output is illustrated as $-->$ and is interpreted as decryption. While $--<$ represents the encryption on the returning path.

Request

1. Single Mix on request

The following method describes the anonymity methods for one Mix of forwarding the messages to the destination:

$K_1^+(R_1, K_a^+(R_0, M), A) --> K_a^+(R_0, M), A.$

The Mix decrypts the input with the private key K_1^-, separates the random string $R1$ and send the remaining content forwards to destination as output.

2. Cascade of Mixes on request

For the cascade, the method remains the same, with the difference that the request is nested in the series of Mixes.

$K_n^+(R_n, K_{n-1}^+(R_{n-1}, \ldots.., K_2^+(R_2, K_1^+(R_1, K_a^+(R_0, M), A)))\ldots) --> \ldots$

$K_{n-1}^+, (R_{n-1}, \ldots, K_2^+(R_2, K_1^+(R_1, K_a^+(R_0, M), A))\ldots) --> K_a^+(R_0, M), A.$

The first Mix decrypts information with K_n^-, keeps the R_n and routes the message to the next Mix. The last Mix forwards the message to the destination, where the responder decrypts it to plain text.

Response

Now the reply needs to be sent back. Since the initiator should remain unknown, the following procedure is responsible for routing the response back to source.

3. Single Mix on response

Back routing for a single mix:

$K_1^+(R_1, A_x), K_x(R_0, M) --< A_x, R_1(K_x(R_0, M)).$

After decrypting the related message part with K_1^-, a mix uses the random string R_1 to re-encrypt the

message. Then the initiator get the response. When the initiator should remain unknown, K_x must be a key chosen by the initiator. Only the initiator knows all random generated strings $R_{1..n}$ and the key K_x, consequently he is the only one who can decrypt the message. Searching the entire output of the first mix for the response with a K_x is how an initiator catches the response and remains anonymous.

4. Cascade of Mixes on response
As for a single Mix, back routing exist also for a Cascade, too.
$K_1(R_1, K_2(R_2, ..., K_{n-1}(R_{n-1}, K_n(R_n, A_x))...)), K_x(R_0, M) - - <$
$K_2(R_2, ..., K_{n-1}(R_{n-1}, K_n(R_n, A_x))...), R_1(K_x(R_0, M)) - - <$
$A_x, R_n(R_{n-1}...R_2(R_1(K_x(R_0, M)))...)$.
Every single Mix encrypts the message with the random string and routes the response to the next Mix, from which it received the random string and the request.

2.3.3 Anonymous-Routing example

Figure 2.4 represents the concept of sender anonymity. Every initiator is informed about the numbers

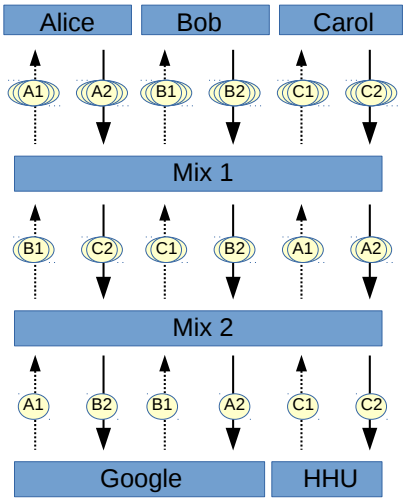


Figure 2.4: Sender anonymity Cascade

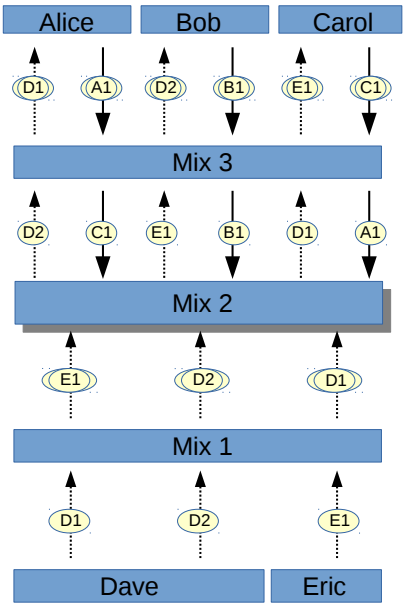


Figure 2.5: Sender and Responder anonymity Cascade

of Mixes it has to go to reach the destination. Equal to the length of the path is the number of encryption of the request as well as the number of random strings. Bob forwards a request to *Mix 1*. There *Mix 1* awaits the requests of all defined initiators, only when all requests arrive at *Mix 1* it forwards messages to *Mix 2*. Note that each requester must send the same amount of messages towards

the Mix. When this is done, *Mix 1* decrypts all requests and exchanges the initiators. Receiving information, *Mix 2* decrypts the requests and mixes the participants again, then transmits them to the destination. The end server i.e. HHU decrypts the content as plain text and then responses. The key selected by Bob encrypts the reply that also contains the random string. Getting the reply, *Mix 2* adds the random string to each reply and routes them to *Mix 1*. There *Mix 1* adds the random string again and sends it to the requester.

The method represented in the Figure 2.4 can be improved to anonymize the responder as well, see Figure 2.5. Therefor a Mix, which maintains the request and the response at once, must exist i.e. *Mix 2*. The requester acts as described in Figure 4. Bob initiates a request with *Mix 2* as destination. The responder encrypts the reply and forwards it to *Mix 1*. Then the replies get mixed and encrypted via *Mix 1* and are sent to *Mix 2*. *Mix 2* merges the request and the response, then sends a reply to *Mix 3* where this one replies the encrypted response, to the initiator.

2.4 Onion Routing

2.4.1 Working principle

The most known application, which uses onion routing method, is called *Tor*. *Tor* allow user to surf on the World Wide Web anonymously. Further, it prevents data traffic analysis. *Tor* service removes user specific information from the request and establishes a path along the service nodes. The service itself basically pretends to be the initiator of the request or rather the last node on the path. Because of the service structure, only the last node is able to see and sent the request toward the destination. Further, this node does not know the origin source of the request, since it does not contain any user specific information, and receives the request always form nodes inside the service. When an answer is incoming, the node route the response back to the previous node. The respond is traveled back along the path until the first node of the service, there the response is sent to the real initiator.

2.4.2 Anonymous-Routing principle

The first router on the path uses a layer of encryptions according to the number of routers it passes on the way to destination. Each router decrypts a layer before routing the request to another node closer to the destination, represented in Figure 2.6. Only there the message is completely decrypted. The response is transmitted back by the path the request was sent.
The exact algorithm contains four steps. Those are *network setup, connection setup, data movement and destruction and cleanup* [5]. To join the Onion-network a client first connects to the *application*

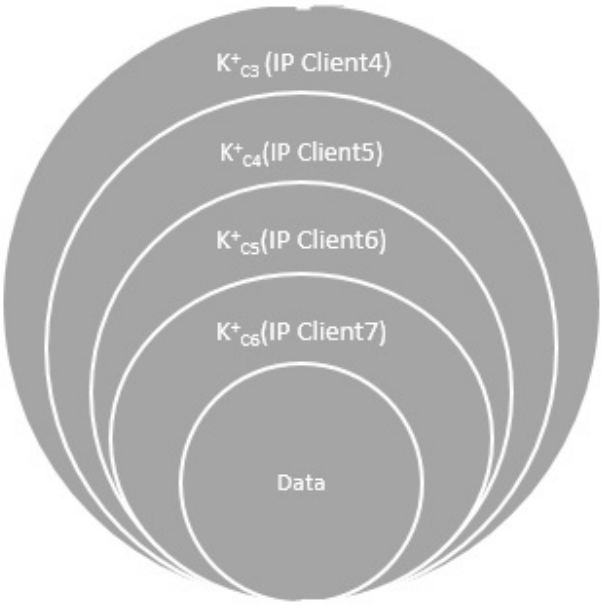


Figure 2.6: Onion

proxy [5]. This proxy removes all information, which could give away the initiator of the message. Only then, the data stream is sent to an anonymous link. An *Onion Proxy* creates a route via the request is going to be sent, additionally it creates an *Onion* [5]. An *Onion*, Figure 2.6, is a data package that is encrypted by several levels, which depending on the path length. Each *Onion Router* [5] decrypts one layer of the *Onion*, finds the next hop and transmits information to it. *Entry Funnel* is defined as the first router on the path and *Exit Funnel* as the last router on the path [5]. Furthermore, each *Entry* and *Exit Funnel* maintains several requests and responses, while having a different path for each one in the onion network. The *Exit Funnel* is responsible for delivering the message to the correct destination. When the responder replies, data exchange occurs. The data is carried on the same route backwards since every *Onion Router* knows it own successor and predecessor. As long as at least one node, which is an *Onion Router*, on the path remains encrypted, the content of the message remains unknown to a eavesdropper too. Further, the data is split into fixed packets. When the transmission is finished or the connection breaks, a destroy command is send to clean up all information on the path.

2.4.3 Anonymous-Routing example

When Alice begin the request, Figure 2.7. She firstly send the message to the *Application proxy*. There the Alice's specific information is removed and the request is send to the *Onion proxy*. The *Onion proxy* defines the path and creates the *Onion*. This is possible, because the *Onion proxy* knows all nodes on the network. Each layer of the *Onion* is encrypted with the public key of the receiving

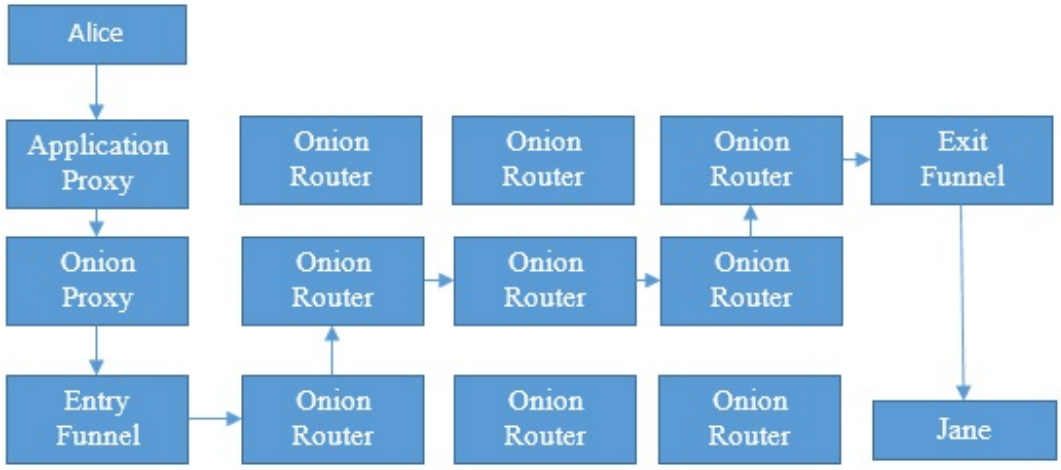


Figure 2.7: Onioin-Routing

node. By decrypting a layer the node finds an IP address, which represent the next hop on the path. On the last hop, *Exit Funnel*, the message gets fully decrypted and the data is transmitted to Jane. When Jane answers, the reply is routed backwards on the same path it was sent.

2.5 Freenet

2.5.1 Working principle

Freenet [7] is a file sharing network in which each node is considered as data storage. For copy right reasons a participant does not know what information is stored on its own node. Each node learns the network consistence by requesting the data itself or by routing it to others. Each data is represented as a number. Similar request numbers are routed via nodes that possess similar information. The more similar requests a node manages, the better it is informed about a certain batch of data. As a result, the routing improves over time. Further, data can be duplicated when requested and deleted if unused, to improve routing performance. The network is adaptive, since nodes with similar data learn each other via intercommunication and maintains the exact location of the data on the network, which also improves routing performance.

2.5.2 Anonymous-Routing principle

Nodes join the network by discovering at least one address of a node inside the network. Each first node that is informed about a new node creates a commitment. This commitment should inform the network about the new node. Since this peer to peer network is unstructured and has no central administration, the only information a node has about the network comes from a locale routing table. The commitment is sent to a randomly chosen node from the routing table. Each commitment has a number of hops, which it may pass before getting invalid, called *hops-to-live* [7]. On this procedure, random nodes learn about the new node and route a request forward, if necessary. For a data request, a user must first calculate the key of the data file before sending a request. A key represents a stored data. Then the user sends the first request to itself to check if its node contains the requested data. If a node does not hold the requested data nor the exact entry of the key in the routing table, the node defines the next hop via the number closest to the requested key. So the request reaches the source of the file. If no requested neighbors contain the requested file, the request is send back to the predecessor. The request is sent back to the real initiator when the *hops-to-live* reached zero. Additionally each request has a unique identifier to prevent routing loops. Loop can occur when node receives the same request more than once. An example of the prevent-loop procedure will be described further in the Figure 2.8. Note that each node recognizes the current predecessor as the initiator of the request so the real initiator remains unknown. When the file is found, it is routed back on the same path it was forwarded. Further, each node on the path creates an entry in the local routing table for the request and copies the file into its own storage. The origin source is sent with the file, to provide more anonymity each node on the path may pretend to be the source too. So when the file arrives at the initiator node, it takes an entry of the last node which claims to be the source, in the local routing table. Those steps can reduces the anonymity degree but are improving instead the routing quality of the network by grouping nodes with same interests. Containing an entry of a key, other nodes will sent similar requests more likely to this node then to any other. By replying the requests, a node gain more information about the network and the storage of other nodes. When a storage of a node reaches the limit, the least recently used data will be deleted to insert the new one.

2.5.3 Anonymous-Routing example

Starting with the *node A* which initiates a request, Figure 2.8. First *A* searches its own storage for the requested file. Not finding the file, *A* creates number of hops-to-live. When these expires the request is aborted. Since the routing table of *A* has the record of *B* as the nearest key to the requested one, *A* chooses to route via *node B*. *B* proceeds on the same strategy and forwards the request to *node C*. There, *C* cannot find the file and is unable to send the request any further, so a response with a failure message is tracked back to *B*. Then *B* searches the routing table for the second closest key,

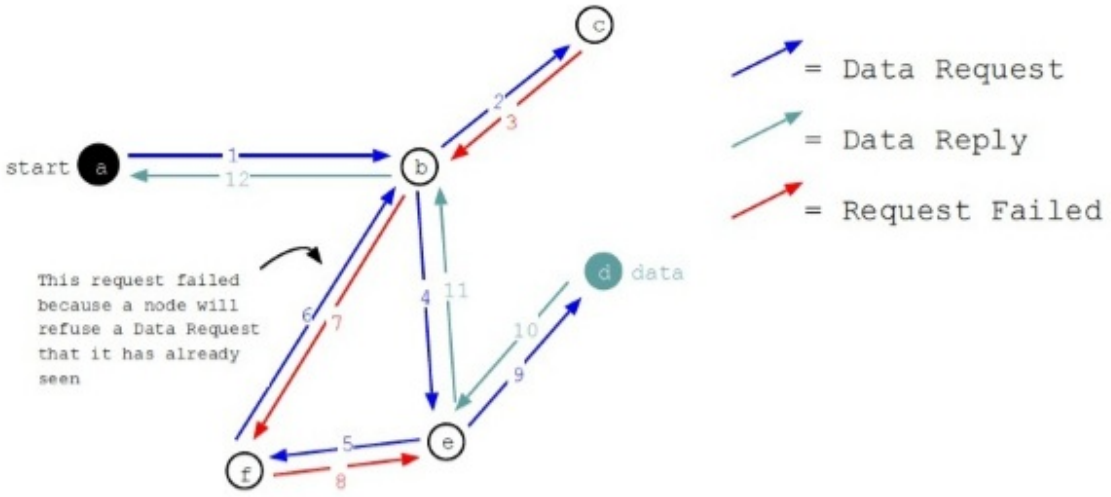


Figure 2.8: Routing on Freenet [7]

which holds the *node E*. From *E*, the request is passed over to *F* where *F* considers *B* as the next hop. Because of the unique request ID, *B* discovers that it already has this request and prevents the routing loop by backtracking the request to *F*, with a failure message. *Node F* behaves as the *node C* and backtracks a failure message, too. When *E* receives the failure message from *F* it forwards the request to *node D* with the second closest key. Finding the file, *D* sends it back to *E* where *E* duplicates the file and routes is back to *B*. From *B*, which duplicate the file as well, the requested file is forwarded to *A*. Additionally *A* saves an entry on the routing table of the last known source of the file, in this case it may be the *node B, D* or *E*.

Chapter 3

Design and Solution

This Chapter represent different anonymity routing approaches i.e. designs for the structured peer-to-peer overlays. First, the content of the third Chapter consider the given circumstances by presenting the given simulator and the structured overlays, where the designs should be implemented. Unfortunately, only the *Chord* [10] overlay implements the two chosen anonymity routing designs. The designs are represented. From "Friend in the Middle" to "Friends" design, the each design considers deeper overlay impact and anonymization aspects as the previous. Then *AChord* [12] an already existing anonymous approach for the Chord overlay, is represented, which I do not design. Although it actually belongs to the Related Work Chapter, AChord is maintained in this Chapter because AChord specifies explicitly on the Chord overlay, other as the represented anonymous routing models in the Related Work Chapter. Further, AChord is implemented and evaluated in this paper, and influence my solution i.e. "SChord". SChord is the second implemented approach on the simulator, after AChord, for the anonymization of the Chord overlay. Afterwards the three designs Chord, AChord and SChord are compared and evaluated in Chapter 5.

3.1 Simulator

To simulate realistic network conditions and achieve proper results the *PeerfactSim* [8] simulator is utilized. This simulator contains different layers to work and evaluate with. Moreover it allows an in depth study of the specific chosen properties, like overlay condition, member information, data exchange and more.
Therefore, *PeerfactSim* maintains six layers: *User Layer*, *Application Layer*, *Services Layer*, *P2P Overlay Layer*, *Transport Layer*, and *Network layer*. Additionally a *Churn Model* exists to simulate the join and leave behavior of the users in the network. The two layers that need to be modified are the *Application Layer* and the *P2P Overlay Layer*. The implementation of the anonymous routing method was planned to be on the *Application Layer* to test and evaluate the anonymity on each structured

overlay implemented in *PeerfactSim*. Unfortunately, this was not possible. The modifications must be done in the routing method inside the overlay, since the focus lies on the anonymity of a peer-to-peer overlay and not in the design of an application, for any overlay, that is not anonymous. Although each overlay has different routing approaches, the designed anonymity method can be transferred to any other structured overlay implemented in *PeerfactSim*. The *Application Layer* is used to investigate the information a user gains of the network, in order to confirm or disprove the functionality of the anonymity service.

3.2 Chord and Pastry

For a lookup the structured overlays use the impact of a DHT to find a node related to the key. Note that each data has a unique key in the DHT. Further, each node is related to a key,too. When a node sends a request for the data, it initially starts a lookup to find the key/node in the DHT. Finding the key it receives the IP Address of the responsible node. Afterwards the routing algorithm of the overlay starts to route the request to the node containing the data. Both overlays, *Chord* [10] and *Pastry* [11], are based on the ring topology. Each node maintains a local list of nodes it has contact to.

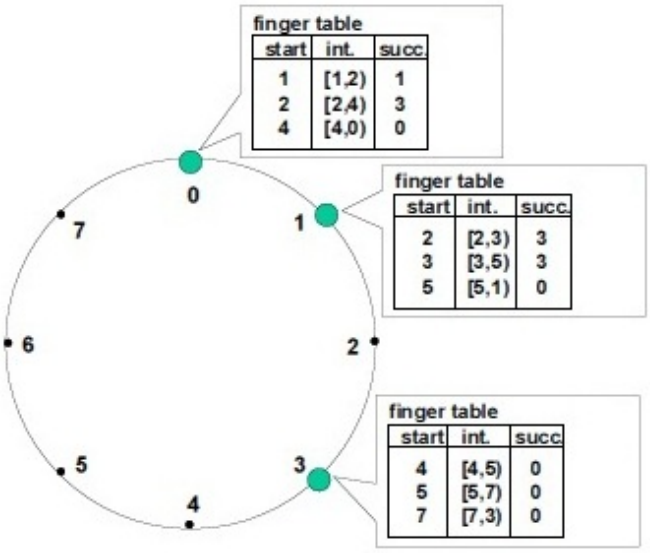


Figure 3.1: Chord Finger Table [10]

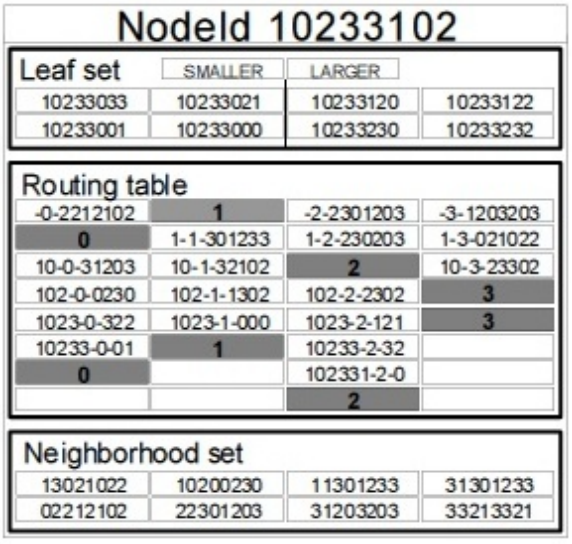
3.2.1 Finger Table in Chord

In Chord it is a list called *finger table* [10] where entries of requested nodes are recorded. The maximal number of nodes each finger table may contains is $O(logN)$. Each node has a predecessor and a

successor, those are defined on the join procedure, to accomplish the ring structure. When a node joins in, it receives the finger table information from the own successor. Then the node periodically updates the finger table. When updating the finger table each node defines an ID for a position in the finger table and search the responsible node for this ID.

3.2.2 Routing in Chord

Further, the nodes in the chord ring are always enumerated and have an ascending order, represented in the Figure 3.1. This "clock" design has affects to the routing. Nodes may route requests only forward to the next higher node. Before the initiation of the routing, a key (which is interpreted as an ID) for the requested data must be calculated. This is necessary because any key is related to a node on the overlay, so one specific node must be responsible for the calculated key. The routing begins when the requested key is generated. The initiator node searches first in its own finger table for a responsible entry of the destination node. When such entry exists, the node send the request directly to the destination node. Otherwise, the node routes the request closer to the destination node. The successor node, when the exact node is not maintained in the finger set, is the next node in the ring.



NodeId 10233102

Leaf set	SMALLER	LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routing table

-0-2212102	1	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	2	10-3-23302
102-0-0230	102-1-1302	102-2-2302	3
1023-0-322	1023-1-000	1023-2-121	3
10233-0-01	1	10233-2-32	
0		102331-2-0	
		2	

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

Figure 3.2: Pastry Table [11]

3.2.3 Routing Table in Pastry

Other than in Chord, Pastry nodes maintain three sub lists the *Leaf Set*, the *Neighborhood Set*, and the *Routing Table* [11]. Figure 3.2 represents an example of those three sub lists. The Leaf Set contains nodes with longest prefix matches to the ID of the node, the half of them lesser and the other half is greater compared to the node ID. The Neighborhood Set maintains geographically closest nodes of the recent node. The Routing Table is responsible for entries of nodes which route information through this node and has appropriate prefix. In general, the Neighborhood Set is not applied for routing but for other Pastry specific purposes.

3.2.4 Routing in Pastry

So beginning with the key a node takes a local lookup to check whether the node itself is responsible for a requested key. When the node is not responsible for the key further revision occurs, otherwise the routing is fulfilled. The node compares first the requested key with the range of the Leaf Set. According to the result, the node routes the request directly to the destination node, which must be a node from the Leaf Set. In the other case, a more similar node closer to the key is chosen, via a lookup to the Routing Table. All nodes proceed on this method until the destination node is found.

3.3 Design

3.3.1 Friend in the Middle

- The first attempt was to hide only the initiator form the responder. This design aims to hide the initiator by replacing it with a chosen friend. The design uses the public key en/decryption to hide the conten of the message from eavesdropper, further it needs a list of nodes where to chose the friend node from. To fulfill the requirements the overlay must provide these two conditions. Therefore, the sender of the message defines a new node for each request, e.g. like a trusted friend, and sends, the ID of the awaiting message and its own node ID. This friend node awaits the response. When the response arrives at the friend node it forwards the answer back, to the initiator of the request. The request or response is always encrypted with the public key of the current destination node, destination nodes are represented in Figure 3.3 as squares.
  The routing of the request starts from the initiator S represented in Figure 3.3, which forwards the request to a random node chosen via the routing table A_1. (In Figure 3.3 each empty arrow form S to R route the same content and the arrows form R to F route the same content.) This

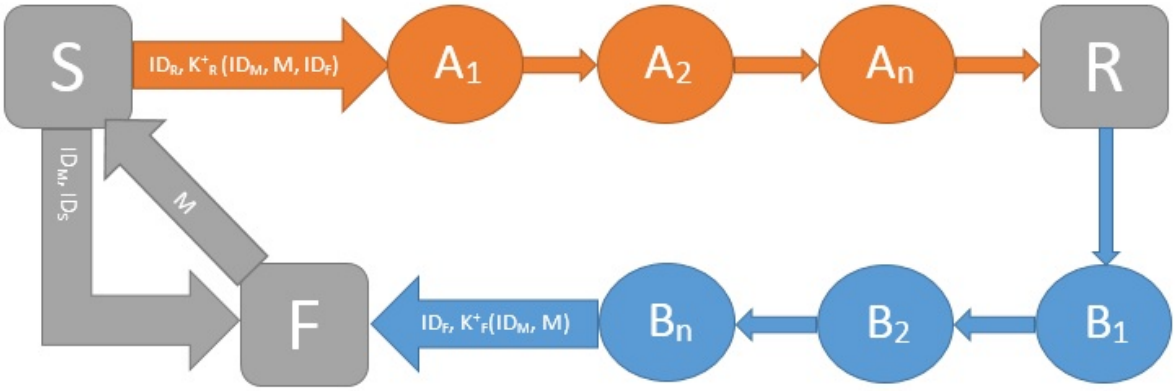


Figure 3.3: Friend in the Middle routing method

random node A_1 proceeds on the same way, by routing the request to another random node A_2. For this procedure, the request forwarding method of crowds can be applied, to randomize the number of next hops before the request is finally routed to the destination. For convenience, lets consider that the third node A_n sends the request directly to the destination. If A_n is unable to send the request directly it will from now on route the request only into the direction of the destination. If a node does not contain the responder in his routing table it routes the request to the next known closest node to the responder ID. When the destination node R (responder) receives the request, it decrypts the message to find the node ID_F and the ID of the message ID_M. The node ID_F defines where the response should be routed to. Again, node R choses a random node to route the response to B_1. B_1 sends the response to B_2. When the B_2 node decides to send the message forward to the destination, any further nodes (if existing) e.g. B_n route the message near to the destination F (friend node), too. Finally, when the trusted node F receives the awaiting message it decrypts the message, checks the message ID_M and sends the response directly to the initiator S.
Although here are some serious disadvantages compared to the next designs, this method has certain unique properties. Nodes on the path do not know whether the forwarded message was the request or the response. The only information seen is the node ID and the node public key. Additionally the friend node changes every time, to gain more anonymity when a particular friend node may be corrupt.

3.3.2 Single User

- Working further on the improvements of the “Friend in the Middle” design, it turned out that the onion routing is the most anonymous and secure method to hide the initiator. Since the initiator itself defines the path it wants to route the message, additionally the initiator encrypts the

message and creates the whole layer model Figure 3.4. According to this procedure, each node is only allowed to see the content of the message that the initiator wants this node to see. In general, eavesdropping of the content and anonymity are two different approaches and must not coexist or be related to each other. However, in this particular design the anonymity is provided by encryption of the content.

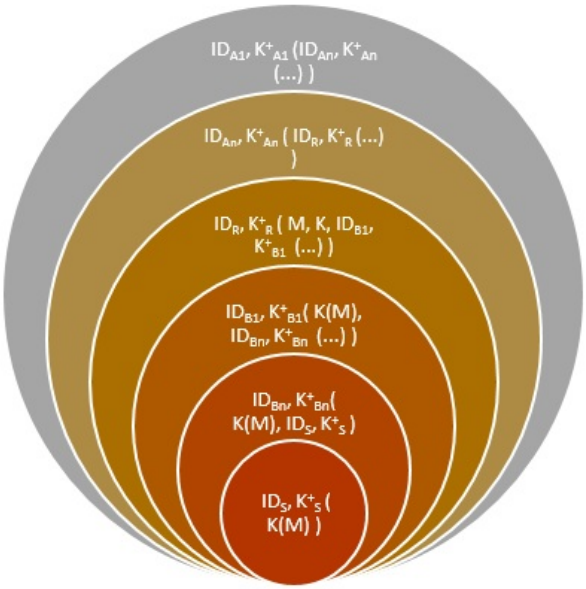


Figure 3.4: Layer Model

Before initiating a request, the sender defines a path, of how the request and the response will be routed. The path contains a number of nodes that are chosen randomly and/or by the routing table. When this is done, the message gets encrypted with the public key of the receiver and additionally contains another key, for the receiver, to encrypt the response as well. Further, any nodes on the route, including the receiver, get encrypted information of the next hop on the path. Figure 3.4 each layer represent the next node and must get the correct information to achieve the goal i.e. reach the initiator with the response message. Only the node related to the current public key can find the next node on the defined way. When the receiver encrypts a layer, it finds a request, a key (a key generated by the initiator) and the ID of the next hop. The receiver encrypts the response with the given key and forwards the information to the next node. Decrypting a layer, a node send the content further to the next hop, if any exists. When the initiator receives the information, it decrypts the response, since it knows the decryption of the generated key and the algorithm terminates.
Figure 3.5 represent the routing process. The initiator S starts the routing. Therefore, it sends the whole layer model to the defined node, which is A_1. There A_1 decrypts a layer to find the next node where to route. This is A_n, now A_n proceeds in a same way and routes the content to R. R is the responder which has the requested information. When receiving the package, R

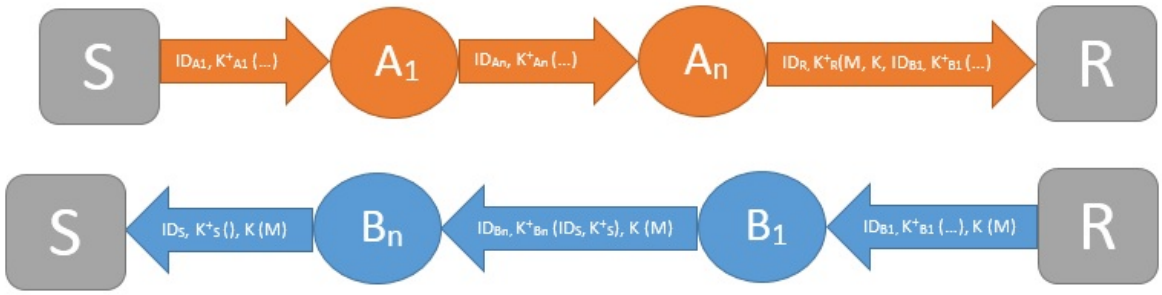


Figure 3.5: Single User routing method

decrypts it to find the content. In this example, it contains the following: the message itself, the key to encrypt the response, next node and the public key of the next node. The only additional operation, which R does, is to encrypt the response message before routing the content to the next node. Since the response cannot be "hidden" on the S layer (because the responder must therefore know the initiator) the response is additionally encrypted with the given key. Although there may be better solutions for the encryption of the response, this method does not affect the anonymity aspect that is why it will not be further considered. As well as all nodes on the path, the responder R sends the response to the next given node B_1 from the decrypted content. B_1 decrypts his layer to find the next node B_n. B_n decrypts his layer too to send the response to S. S on his part decrypts his layer and the response message. Here B_n can not distinguish if S is the initiator or the next node on the path. To provide more guaranty S can create a dummy node for itself to fake the size of the next layer and so distract the B_n node.
Although this method is extremely flexible for the initiator of the request, this method provides no anonymity to the responder of the request. In a peer-to-peer overlay each member represents a node. Other than in "Tor" where the request is sent to the web server, the request and response of particular information occurs by the nodes inside the overlay, since no web servers exist. While each node of the certain overlay must remain anonymous, the method must guarantee the anonymity of the responder, too. In fact, this method does not provide the necessary functionality of hiding the responder.

3.3.3 Friends

- To achieve anonymity, the responder must remain unknown to the initiator as well as to the nodes along the path. In this specific overlay type, the only possible way to hide the responder is given through indirect communication. The following method represents an approach for this.
  Figure 3.6 represents the basic idea of this design. Each node defines friend nodes whereby

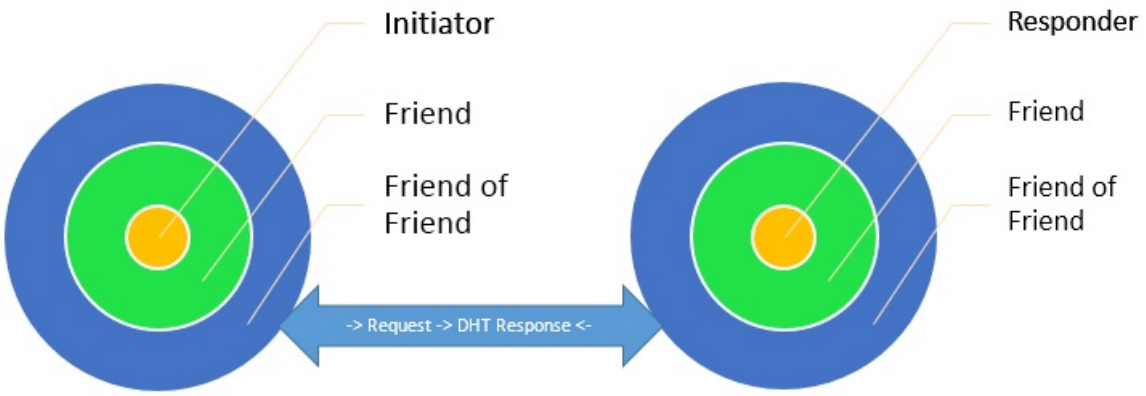


Figure 3.6: Basic Friends routing method

the routing proceeds, the requested information is routed to the responsible node via its own friends. When the responsible node is found the response is routed backwards on the same path is was send. Each node maintains a list of *Friends*, those are nodes that have access to all information inside the friend node e.g. *Responder*. Therefore, each node actually maintains a DHT list of all "stored" information inside its friend nodes. By this means, a node stores its own information plus all information of its friend's storages. The routing begins when the initiator creates a request. The first sender defines a number of friends, which the request must pass before it is forwarded to the destination. When the node is permitted to route the request into the direction of the destination, it takes a lookup on the local DHT. When a node from the DHT is responsible for the requested information the request is forwarded to it, otherwise the request is sent to the next closest node. When a node is responsible for the information, it is unknown whether the node itself contains the information, or it has a friend that holds the requested information. The responsible node has two assignments. First, to protect the real holder of the information. Second to improve the routing of the request. When the friend receives the request, it forwards it to the next friend of whom he "thinks" (according to the DHT) it might hold the information. When several candidates exist, the node forwards the request to all candidates. Each friend forwards the request until the responder is found. The routing loop prevention as well as the routing respond occurs via the request ID. Each request possesses a unique ID, when a node receives a double request ID, the last request is either ignored or the node receives a failure message. The response must contain the request ID to route the answer backwards to the initiator. Further, it must be considered that deadlocks are possible. To avoid this behavior and reduce unnecessary traffic on the network, friend nodes route the request, if they are the initiators, directly to the next node, which may have the requested information. This will not affect the algorithm since it is the usual procedure for any friend. Additionally a friend is not aware whether his friend is the original responder or not.

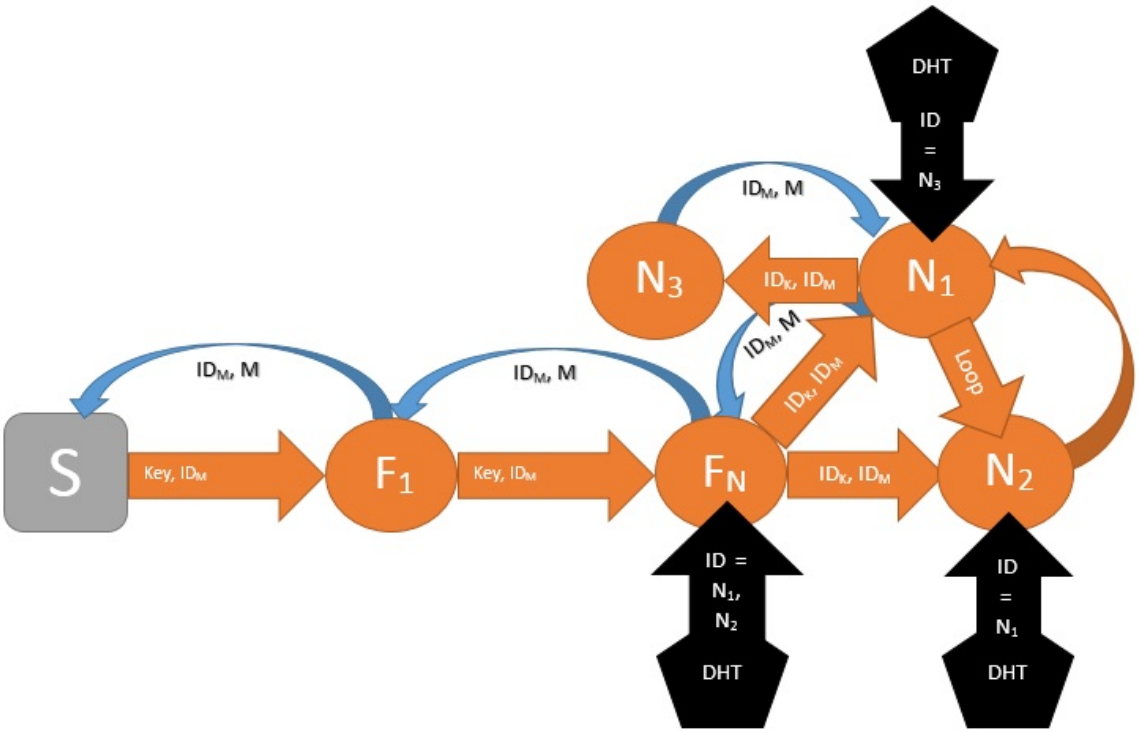


Figure 3.7: Advanced Friends routing method

Figure 3.7 shows the routing procedure. In the given example nodes F_1 to F_N are the friend-nodes chosen by each next friend-node. F_N initiates the search process for the request. Therefore, it firstly looks into its own storage. If the node itself does not contain requested information it searches the DHT for the node, which is responsible for the information. Nodes maintained in the DHT are always friends. In this example, those are N_1 and N_2. F_N sends to both of them a request. When N_2 searches for the ID, it define the next node as N_1. N_1 has received a request from F_N and then from N_2 with the same ID so the second request is returned back as a loop. Now N_1 look for the next node since it is does not contain the requested information. From N_1 the request is send to N_3, this is the real holder of the information. From there on the response is routed back via $N_3 -> N_1 -> F_N -> F_1 -> S$. Each member of this chain assumes that the node before is the responsible node for the requested information.
This method may turn out to be good for the anonymity of each node but the implementation of this method is not possible without changes of the existing overlay in depth.

3.3.4 AChord [12]

- Because the represented solution did not completing the required purpose or could be not adapted into the simulator an existing anonymous routing method *AChord* [12] has been sug-

gested. This method relates only to the Chord overlay and was not evaluated in the given paper. The authors writes that the lookup must be related to the hash value of the desired object and not to the node that is responsible for its key [12]. Further, in the returning of the response messages, the responsible node should not be mentioned. When a new node joins the network, it must classify itself between the predecessor and the successor nodes. Therefore, the ID of the successor and the predecessor are needed. AChord limits the access by allowing a node only to search for the own predecessor and successor. Because of this, the authors supposes that only the iterative search process is possible during the joining operation. Only when nodes leave the overlay, the nodes are allowed to gain a new predecessor or successor. According to this step, the new predecessor or successor must confirm the requesting node as his predecessor or successor. Rather than in Chord, in AChord on a data request, which is recursive, the node always gets a key instead of an ID back. Additionally, the response is tunneled back to the initiator. A main part of Chord is the finger table where the IDs of contact nodes are maintained. The Chord ring contains enumerated amount of all nodes, the finger table tries periodically to update itself to find the best set of nodes for its own finger table to minimize the routing procedure. The request for the update occurs recursively and is routed via all possible nodes towards the best node. This procedure is replaced in the AChord solution with the following. When the node searches for the i'th position for the own finger table it communicates directly with the node on this position. Then the requested node N' ask its own finger table for the node on the i'th position. While the node N' knows the ID of i'th position it determinates which node is the best match for the reuester position. Rather the node itself or the node on i'th position, then the better node is returned to the requester.
This approach will be considered further in the evaluation chapter.

3.4 Solution

The final solution, called SChord, is based on the AChord approach but is modified by me on some aspects. SChord as well as AChord solution are both implemented into the simulator to illustrate and evaluate the differences between them, and to point out the described improvements.

3.4.1 Join procedure

The joining procedure is changed because the iterative AChord search may reveal too much information of the ring structure to the joining node, especially when the ring is steady. This may be a thread of because each node communicates directly with the joining node. As described before each joining node must know the ID i.e. IP of the successor and the predecessor, so the joining must compare those

IDs too. On this process, the joining node can record all the nodes it communicates with. The join process is recursive in SChord, although each node on the routing way is aware of the new joining node (as well as in the iterative method). The difference is that the newly node will almost not gather any information about the network on joining. On the other side, the information about the new node is mostly not useful to other nodes, since it is unknown where the node will be placed in the network and to whom it is connected. Further, when the successor node is found it communicates directly with the new node, so no other node knows the exact relation between those nodes (except the predecessor). So nodes on the routing path gain no information about the network since they never receive any information back. Direct communication is allowed while each node in the overlay knows the ID of its direct neighbors (predecessor and successor). When a node leaves the network, it must inform the neighbors so that they can interact with each other to close the gap and reestablish the ring again. In addition, each node asks periodically its neighbors if they are present. When a neighbor does not respond, the joining method is called again. The leave process of the nodes is not considered in this bachelor thesis.

3.4.2 Routing procedure

Chord is based on a ring topology, further it is structured and routes the request only in clockwise direction. To establish a certain proved degree of anonymity the requester and the responder must remain unknown to each other. Therefore, AChord tunnels the response back to the initiator. The tunneling procedure is not clearly described in the paper [12], so it was interpreted as the backward routing via the same nodes the request was send. For each node, only the direct neighbors and the nodes from the finger table are known. According to this knowledge, each request reaches the target in maximal $N-1$ hops, if routed only per successor. The same statement is valid for the (backwards) routing of response as well.

3.4.3 Finger Table

The finger table entry contains the perfect ID for each position in the finger table, according to this ID a responsible node is recoded and returned for this position. It may be possible to exchange the returned node ID with the requested key ID to hide the responsible node. Alternatively, the finger table may be completely removed from the Chord routing method, since this would guarantee a certain degree the anonymity. Of course this would affect the performance but that is a general problem of the trade off between performance and anonymity. Further, it must be evaluated how much information the nodes can record from entrys of the finger table in AChord and SChord. That is why the finger table for now remains implemented. For the updating of the finger table AChord provides a curiously method

of checking the nodes i'th position. The given method is questionable, since it rarely would give any other node back than the one which is now in the finger table for this position. This method is slightly modified in SChord as following. The node which updating the own finger table, request the node that is already on the i'th position with the perfect key for this position. This i'th node take a look up firstly to itself if it is responsible for this key. If the node is responsible, it returns itself as the responsible node. If not, the node searches for the next closes responsible node to the key and return this node's ID instead. Each finger table has the size $O(logN)$, where N is the number of nodes on the network i.e. the network size. Additionally, each node allows to have entries of with the maximal distance to the node by $N/2$. By knowing this information, members are able to analyze the node's range of responsibility for the key. E.g. when knowing node X's and nodes Y's ID or key, it is possible to determinate whether those nodes are neighbors, according to the key or ID and by knowing the network size from finger table. And further define by this the responsibility dimension for the keys of a particular node.

3.4.4 Summary

To sum it up, the danger of detecting the node identity has three potential threads: the join procedure, the routing procedure and in the updating of the finger table. Those processes has to be modified, the next chapter describes the implemented adaptions in detail, where chapter 5 evaluate those adaptions and compare the Chord, AChord and SChord approaches on the anonymization aspects.

Chapter 4

Implementation

This chapter described the already implemented *Chord* [10] design and the changes which have been made to achieve the required anonymity conditions of *AChord* [12] and SChord [1]. Further, it describes the spy procedure. To test these anonymous conditions each service is modified by a spy operation. The spy operation can act passively or/and actively, to differentiate between the types of attack and to achieve proper evaluation results. Each service Chord, AChord and SChord is tested by active and passive type of eavesdropper, and the results of the spy process are compared on these three services in the Evaluation Chapter.

4.1 Chord

When a user defines an operation it is performed from the *AnonymityApp* class. This class contains the operations, which a user is allowed to define in the configuration file. Those operations are *get*, *store*, *initApp* (join) and *spy*. The flowchart in Figure 4.1 represents the detailed view of the *get* function and the *overlayNodeLookup* procedure. In the given Chord version all four operation *get*, *store*, *initApp* (join) and *spy* perform the search via the *overlayNodeLookup*. This implies that each operation type is returning a node back to the requester, before the actual condition is performed, Figure 4.2. Further, the *UpdateFingerPointOperation* uses the *overlayNodeLookup* for the updates of the finger table as well. The communication with the other nodes is defined through the *send-function*, in fact there are two send functionalities the *sendAndWait-* and the *send-function*. Referring to the name of the function, *sendAndWait* awaits a response back where *send* only forwards the message to the defined node. Additionally *sendAndWait* has a timer $MESSAGE\_TIMEOUT$, when this expires before the response arrives, it tries to resend the message again. If a certain number of messages has been sent without success i.e. no response is received, the operation is aborted and terminates with a failure condition. To differ between the awaiting messages, a *receivingEvent* exists. According to this

[1] The SChord is an abbreviation for the defined solution of the Chord-Overlay represented in Chapter 3

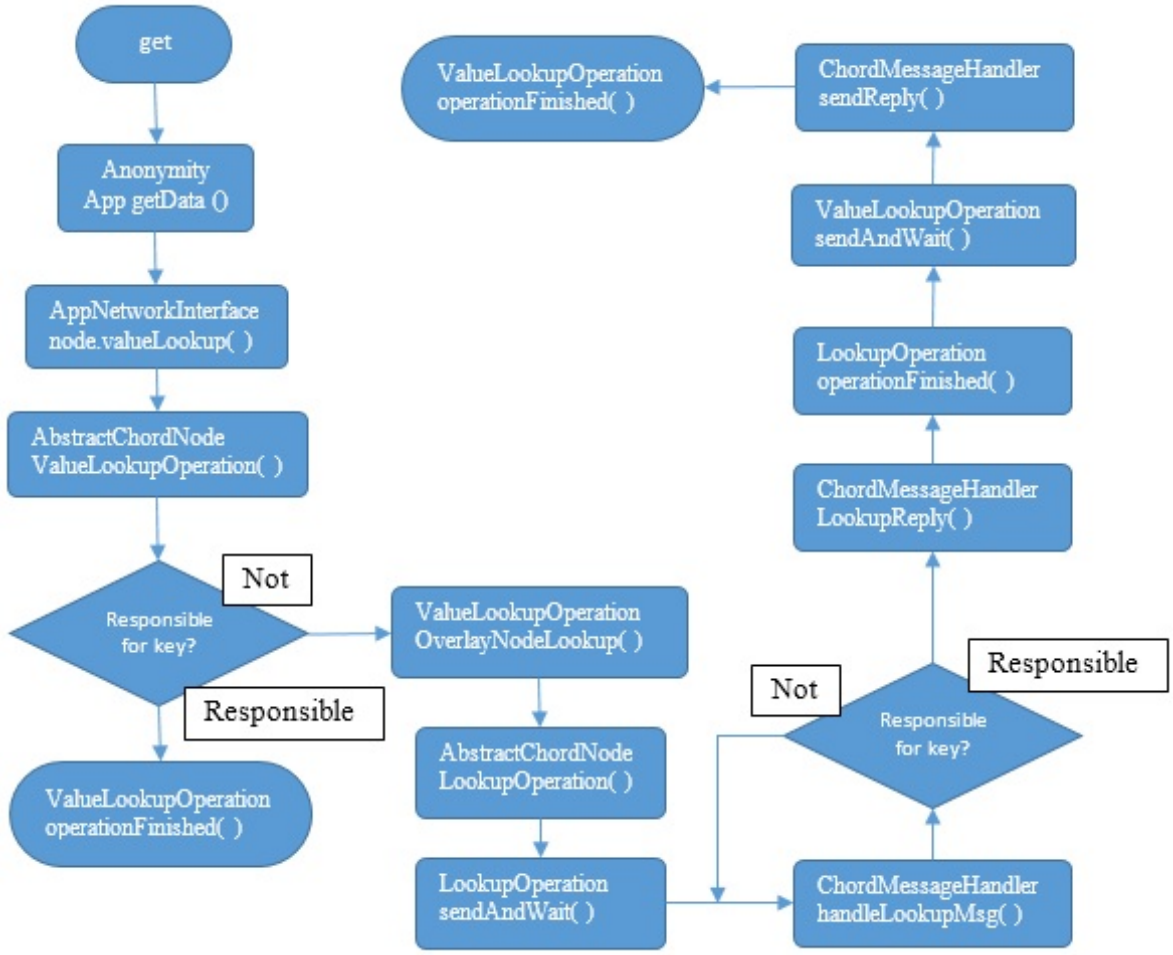


Figure 4.1: Flowchart of the *get* operation in Chord

event, the node that sends a request should get the correct response to its request. Nevertheless, this functionality does not work properly when many nodes intercommunicate at the same time, because the response does arrive at the wrong requested node. The second functionality is the *send* function, which forwards the message only once to the destination, without cornering about package loss. To distinguish between different operation types, each message must be defined into a type. The sent messages arrives usually on the ChordMessageHandler class, if the message is a reply message other classes can fetch it, referring to the *receivingEvent*, see Figure 4.1 when the node is found. So when the message arrives at the *ChordMessageHandler*, the type of the message is first checked. Then the *ChordMessageHandler* executes the defined properties for this specific message type, e.g. the search. The search i.e. loop occurs when the *ChordMessageHandler* is not responsible for the key, see Figure 4.1 down right ChordMessageHandler, the request is then send to another node's *ChordMessageHandler*, this process repeats until the responsible node is found. In general, the *ChordMessageHandler* has the task to search for the responsible node of the request. When found the responsible node a

response is sent back to the requester. There the defined operation is finally executed. The requester receives firstly the ID of the responsible node, then the requester send again the key to the responsible node to get the information from it, as can be seen in Figure 4.2.

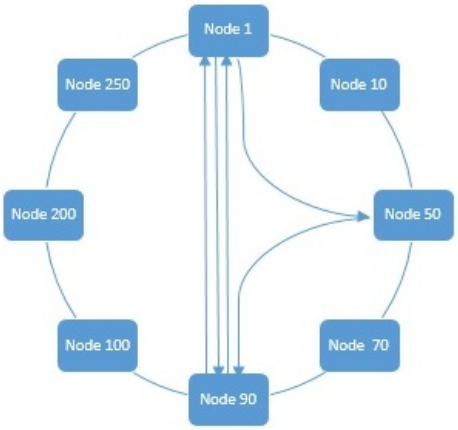


Figure 4.2: Chord request routing

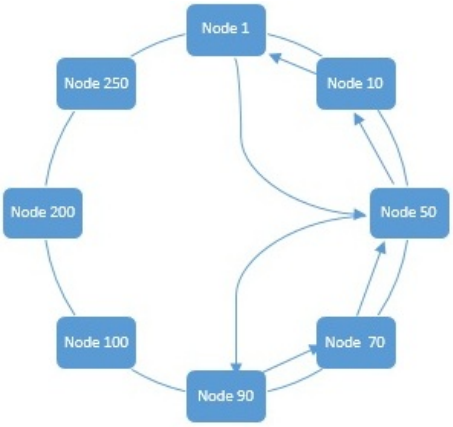


Figure 4.3: AChord and SChord request routing

4.2 AChord & SChord

4.2.1 Routing procedure

This routing behavior of Chord, see Figure 4.2, has to be changed in *AChord* [12] and SChord, my adapted design, to achieve the anonymous routing. Therefore, the *ValueLookupOperation* (*get*) and the *StoreOperation* (*store*) do not receive nodes back but only the needed information. Still to route the request and response, the node must know the ID of nodes it communicates with. According to the AChord paper the response has to be tunneled back. The tunneling procedure is unfortunately not exactly described in this paper. The implementation was supposed to route the request via the information from the finger table and the response backwards via the same nodes. Because of to much awaiting responses, (while each node on the way pretends to be the initiator of the request) the response by the *receivingEvent* hit the wrong nodes. The anonymous routing procedure was implemented with the following changes. The request is still sent towards the information form the finger table, but the response routing occurred only by the predecessor, Figure 4.3. So each predecessor pretends to be the responsible node for the given operation.

4.2.2 Join procedure

The *join*-operation of Chord is implemented recursively, although the SChord *join*-operation behaves similar to the one of Chord it was re-implemented. The routing of the *join*-operation of SChord and Chord is represented in Figure 4.2. The only necessary change was in the AChord version, since it has to be iterative. When the *JoinOperation* is called, the node defines firstly a random next hop to begin the routing. Then the *join*-message is sent to this next hop. The next hop receives the *join*-message by the *ChordMessageHandler* and directly returns the new next hop to the requester. This process is executed as long as the successor is not found. Afterwards, the successor, the predecessor, and the joining node exchange information.

4.2.3 Finger Table

The finger table must be limited to decrease the knowledge of the network and provide more anonymity. When the *UpdateFingerPointOperation* is executed, the *ChordMessageHandler* of the next node (the node that is requested by the initiator) looks at the i'th position of its own finger table entry. It compares then which node is closer, the node itself (next node) or the node in i'th position, to the requested key of the initiator and returns then the closer node. SChord's design limits the finger table as well but is different to AChord's approach. It allows the next node (the node that is requested by the initiator) to check the whole finger table for a node which has the responsibility for the given key, if this node (next node) is not responsible for the given key. AChord's and SChord's *UpdateFingerPointOperation* is iterative since it communicates with one node only.

4.3 Spy procedure

The spy procedure has the task to collect the ID of nodes and if possible to define the responsibility range of the known nodes. The spy procedure is devided in the passive and active process. The passive process saves all nodes which route or request the spy-node, while active process starts a request and tries to determenate the responsible node for the requested key, and by this the responsibility range of the node.

The *spy*-operation can be called passively and actively, and is new to all three Chord designs. The passive spy is executed on the *JoinOperation* via *initApp(true)* function, which define the node as a spy-node. A spy-node records any nodes that route information through. This passive operation affects the *ChordMessageHandler* since this class represent the searching process. The passive spy, eavesdropping on the *join-, store-, get- and update*-messages. The active spy process is called by the *spy*

function and is executed firstly in the *AbstractChordNode* class, then periodically in the *ValueLookup-Operation* and the *StoreOperation*. The requested key changes every time the *spy* is executed on *ValueLookupOperation* and *StoreOperation*. This is a necessary condition to reach new nodes. The defined next hop in AChord and SChord is considered as the responsible node, because the received response is always from the own successor, as can be seen in Figure 4.3. In the Chord design the active spy collects the information from the receiving response, when the object or null (on *store*) or when an acknowledgment (on *get*) is returned. The new nodes from active and passive spy observation of the process are collected in a list. Additionally, each four minutes the *Information* function is called, there the list gets sorted. Then by *SafeToFile* function a file with the collected information is created in the folder of the spy-node. This file contains the known nodes e.g. the network structure and other useful spy information.

Chapter 5

Evaluation

The overlay design has three weak point where the anonymity is threatened. Those are the joining process, the *finger table* [10] update and the routing process. The joining process and the finger table are specified by the overlay, while the routing process may be adapted in some ways. The routing itself contains further three anonymity aspects, sender anonymity, responder anonymity and the intercommunication anonymity between the sender and responder. The implemented spy opeartion considers those anonymity aspects by collecting specific information of the network. This information is represented in the tables of each example and show different aspects of the spy procedure and techniques, example deals with. Those examples aims to point out the different elements of the Chord overlay and to evaluate the implemented suggestions designed in this thesis and the following paper [12].
The setup for each Chord variant contains for each one example the same properties. The first example has the focus on the anonymous routing procedure (anonymous intercommunication) of AChord and SChord [1] and represent the differences to the origin Chord routing trough the active spying process. While the second example reviewing the finger table impact(sender anonymity, receiver anonymit) and the anonymity affect by joining procedure based on the structure changes. The last example shows the global impact on the anonymity when multiple nodes spying together.

The following operation are used during the evaluation.

- initApp: This operation represents the join process of a node. The values true and false are interpreted as the permission for the spying. False if the node is not allowed to spy. True if the node is allowed to spy. This spy is a passive spy-operation, which only eavesdrop when something was routed by.

- get: The get-operation take a request via an given key and receives an response i.e. object back if such exist.

[1] The SChord is an abbreviation for the defined solution of the Chord-Overlay represented in Chapter 3 and 4.

- store: The store-operation stores an object on the responsible node which is defined by the key.
- spy: The spy-operation is the active spy procedure, which contains the get- and store-operations and invokes them periodically.

5.1 First example: Active spy process

For the first example, the properties are set as following:
The simulation time is set to 60 minutes.
This simulation contains 514 nodes, where two of them are spy-nodes.
LatinAmerica contains one spy-node, Europe contains 512 nodes and NorthAmerica one spy-node.

Executed operations:

Group name	Time	Operation
Europe	1m-20m	initApp false
LatinAmerica	25m	initApp true
LatinAmerica	30m	spy
NorthAmerica	32m	initApp true
NorthAmerica	35m	spy

Table 5.1: Example 1: Setup for the active spy procedure.

The first column of Table 5.1 represents the geographical position of a certain group of nodes i.e. members. The second column shows the time when the operation begins, if an interval is given it meant the node can decide when it executes the operation. The third and last column is the operation column, it represents the operation provided via the whole group.

Spy Time	0:27:0	0:31:0	0:35:0	0:39:0	0:43:0	0:47:0	0:51:0	0:55:0	0:59:0
Chord	17/513	34/513	37/514	39/514	41/514	43/514	45/514	47/514	49/514
AChord	37/513	37/513	38/514	38/514	38/514	38/514	38/514	38/514	38/514
SChord	6/513	7/513	7/514	7/514	7/514	7/514	7/514	7/514	7/514

Table 5.2: Example 1: Node 192.76.21.1 from Latin America joined on 25m

Spy Time	0:34:0	0:38:0	0:42:0	0:46:0	0:50:0	0:54:0	0:58:0
Chord	23/514	53/514	58/514	60/514	62/514	64/514	66/514
AChord	297/514	297/514	297/514	297/514	297/514	297/514	297/514
SChord	8/514	8/514	8/514	8/514	8/514	8/514	8/514

Table 5.3: Example 1: Node 64.167.127.97 from North America joined on 32m

When a node joins as the spy, it automatically begins the passive spying i.e. eavesdrop. To show the active spying event in the first example, the spy-nodes are acting actively. They *store* and *request* information on the network, while the other nodes do not do anything. In each time interval between the spy outputs, represented in Table 5.2 and Table 5.3 as Spy Time, each spy-node performs one *store* and one *get* operation. The keys for those operations are set to reach respectively another node. The results from the Tables shows the joining procedure, the update of the finger table and the active spy procedure. The two first properties are related to Chord's structure and will be described in the next examples. So let us consider the active spy process, which occurs after the join and updating part. The node 192.76.21.1 Table 5.2 and the node 64.167.127.97 Table 5.3 represents those. The active Chord spy learns about other nodes on each request if it communicates with unknown nodes. Following the time line from Tables 5.2 and 5.3 a Chord node learns in each time interval two new nodes, since it performs two operation. It may appear small sized but it is only a demonstration of the anonymity issue. If this node performs N operation with N different keys, where N is the number of all keys contained in the network, it will know the whole network size and each responsibility range of each node. The active AChord and SChord spy-node does not learn about any new node on the *store* and the *get* operation even if the responsible node is unknown to it. This happens because the response is tunneled back to the requester by the nodes it already knows. The time line in the Tables 5.2 and 5.3 shows that AChord as well as SChord do not learn any nodes from active spy process. The next table reveals the detailed view to the given statement.

Spy Time	0:27:0	0:31:0	0:35:0	0:39:0	0:43:0	0:47:0	0:51:0	0:55:0	0:59:0
Chord known nodes	17	34	37	39	41	43	45	47	49
Chord nodes by join	4	4	4	4	4	4	4	4	4
Chord nodes by FT	5	5	5	5	5	5	5	5	5
Chord nodes by spy	8	25	28	30	32	34	36	38	40
AChord known nodes	37	37	38	38	38	38	38	38	38
AChord nodes by join	36	36	36	36	36	36	36	36	36
AChord nodes by FT	1	1	1	1	1	1	1	1	1
AChord nodes by spy	0	0	1	1	1	1	1	1	1
SChord known nodes	6	7	7	7	7	7	7	7	7
SChord nodes by join	4	4	4	4	4	4	4	4	4
SChord nodes by FT	2	3	3	3	3	3	3	3	3
SChord nodes by spy	0	0	0	0	0	0	0	0	0

Table 5.4: Example 1: Detailed view of the known nodes from Node 64.167.127.97

Each result (number of known nodes) is subdivided in the three categories see Table 5.4, join procedure, finger table, spy procedure. This subdivision allows a detailed evaluation for the above existing results. Although the given result do not differentiate between active and passive spy process the setup of the simulation allow to point out the exact process this example concentrates on. When a node joins

the network, it needs some time to update the finger table properly that is why in the first two results, according to time line, stabs out. Table 5.4 shows that the active Chord spy-node continuously gain new infoarmtion about the network while acting actively, since no other procedures are executed. While the AChord and SChord spy-node does not get any new inforamtions after the updating/joining process. According to this AChord and SChord know 0% of the requested-responsible nodes,while the Chord knows 100% of the requested-responsible nodes. Again, in the given example represents that no new node can be learn from active spy process in the AChord and SChord. To improve the performance of routing, the finger table define the next hop for a given request, actually the finger table determinate the routing process. The next example describes the differences on the finger table as well as on join process of the three Chord versions.

5.2 Second example: Join process and finger table impact

For the second example, the properties are set as following:
The simulations time is set to 120 minutes.
This simulation contains 1015 nodes, where three of them are spy-nodes.
NorthAmerica contains 712 nodes, LatinAmerica and France has each 150 nodes, Australia, Germany and Italy represent each, one spy-node.

Executed operations:

Group name	Time	Operation
NorthAmerica	1m-30m	initApp false
Australia	15m	initApp true
LatinAmerica	32m-45m	initApp false
Germany	50m	initApp true
LatinAmerica	55m-70m	store
France	75m-85m	initApp false
France	90m-100m	get
Italy	101m	initApp true
France	105m-114m	get
Australia	115m	spy
Germany	115m	spy

Table 5.5: Example 2: Setup for the passive spy / finger table and joining procedure.

The given results[2] show that the join operation of the AChord allow gaining more information, ac-

[2]Keep in mind that the network may need time to take effects and reestablish the ring structure.

Event	1-50m *join* of 864 nodes		55-70m *store* by 150 nodes		75-85m *join* of 150 nodes		90-114m *get* x2 by 150 & one *join*		
Spy Time	0:17:0	0:57:0	1:1:0	1:5:0	1:17:0	1:29:0	1:33:0	1:45:0	1:57:0
Chord	16/393	61/864	62/864	65/864	65/894	79/1014	79/1014	79/1015	79/1015
AChord	22/391	27/864	27/864	27/864	27/872	41/1014	41/1014	41/1015	41/1015
SChord	19/392	33/864	33/864	33/864	33/887	47/1014	47/1014	47/1015	47/1015

Table 5.6: Example 2: Node 61.8.4.98 from Australia joined on 15m

Event	1-50m *join* of 864 nodes		55-70m *store* by 150 nodes		75-85m *join* of 150 nodes		90-114m *get* x2 by 150 & one *join*		
Spy Time	0:52:0	0:56:0	1:0:0	1:4:0	1:16:0	1:28:0	1:32:0	1:44:0	1:56:0
Chord	17/864	37/864	41/864	41/864	41/879	50/1014	50/1014	50/1015	50/1015
AChord	15/864	15/864	15/864	15/864	15/868	15/1014	15/1014	15/1015	15/1015
SChord	7/864	8/864	8/864	8/864	8/870	8/1014	8/1014	8/1015	8/1015

Table 5.7: Example 2: Node 205.153.101.1 from Germany joined on 50m

Event	1-50m *join* of 864 nodes		55-70m *store* by 150 nodes		75-85m *join* of 150 nodes		90-114m *get* x2 by 150 & one *join*		
Spy Time	-	-	-	-	-	-	1:43:0	1:47:0	1:59:0
Chord	-	-	-	-	-	-	21/1015	34/1015	34/1015
AChord	-	-	-	-	-	-	178/1015	178/1015	178/1015
SChord	-	-	-	-	-	-	6/1015	6/1015	6/1015

Table 5.8: Example 2: Node 210.185.12.33 from Italy joined on 101m

cording to network size, than the Chord and SChord on join operation. Therefore, look at Table 5.8 and Table 5.3 from first example, at the first existing entry. This is where the AChord node knows from begin on a relatively huge amount of nodes. There may be exceptions of course, since join process is executed onto a randomly node, so if the random node is placed closely to the successor the joining node will not know much about the network, Table 5.7 shows this. Still the most affect from the join has the AChord version, since the process occurs iteratively, what means that the new node communicates directly with each node on the way toward to successor. Because of this direct type of communication, the new node is allowed to record all necessary information about the (next) node it communicates with. Additionally the imprecise finger table information boost this impact. Since the nodes do not have enough information about the nodes of the overlay, see the finger table lines of Table 5.4 and 5.9, the new node must route through far more unknown nodes to reach the destination i.e. successor in AChord than in Chord or SChord.

Because the finger table influenced the routing, no additional information can be collected through the passive spy procedure, when the structure is once updated. This is a fact, since the routing refer the information from the finger table. See the last event column (in time from 1:45 to 2:00) on Table 5.6

Spy Time	1:43:0	1:47:0	1:51:0	1:55:0	1:59:0
Chord known nodes	21	34	34	34	34
Chord join	4	4	4	4	4
Chord FT	5	5	5	5	5
Chord spy	12	25	25	25	25
AChord known nodes	178	178	178	178	178
AChord join	177	177	177	177	177
AChord FT	1	1	1	1	1
AChord spy	0	0	0	0	0
SChord known nodes	6	6	6	6	6
SChord join	4	4	4	4	4
SChord FT	2	2	2	2	2
SChord spy	0	0	0	0	0

Table 5.9: Example 2: Detailed view of the known nodes from Node 210.185.12.33

and 5.7 the number of known nodes does not change although the nodes communicate. In the Table 5.8 on the same time from 1:43 to 1:47 a peak occur because this node joins there. The peak has the most effect to the origin Chord because of the finger table properties. This peak shows the reestablishing process of the Chord ring by other nodes via this node. In other words, when the node has joined the network other nodes may add this newly node to their routing table. This is why there is always a peak in the beginning(when the node join). For a closer look, let us consider the detailed view of the known nodes by the node 210.185.12.33 Table 5.9. As said before the nodes do not learn new nodes by passive spy of the routing procedure. This can be clearly seen by the AChord and SChord the node. When join on a steady structure, node can not collect much information from the existing network not by finger table or by spying process Table 5.9 (Spy- & FT-lines). The disadvantage of the AChord join process was mentioned before. The reason why the Chord node gain more information lies in the structure of updating of the finger table. The Chord nodes can access any node of the length $N/2$ from its own position, in one-step, because each node route the request to the responsible node for the given key. The SChord limits this condition, by allowing each node only to ask the given node if this one has a better match for the key. Consequently, some sectors, which can be seen by nodes in Chord, remains for the same nodes in SChord unknown. The AChord finger table is very imprecise because of the given condition it and does rarely know any new node from the finger table. Again, this passive spy result, since no active operation are performed, is the result of the reestablishing the ring structure and redefining of the finger table. Such peak can be seen on each joining node, and is represented in the each table in the first two intervals according to the joining time. Look at any node Table if you want to.

So only three spots can give away new information about the node. Those are the finger table, the active process executed by the node and the join procedure, which additionally affect the finger table. In Chord all three possibilities reveal new information about the network i.e. ring structure. AChord disable the gathering of the information on the active processes, further it limit the finger table in-

formation to minimum. As one can see in the Tables 5.4 and 5.9, the AChord finger table maintains the minimal number of node in comparison to the other two Chord variants. This limitation implies a potential threat. Because the finger table rarely get a new node, it remains imprecise. As a result, the routing process may not work properly. In the best case, it decreases the performance speed, since the node do not exactly know where to route the request to. The performance loss can be seen on Table 5.6 at 1:17 and at Table 5.7 at 1:16. In the worst case, the routing process initiates a loop where the request never reaches the responsible node. Another aspect is that the nodes, which are maintained in the finger table, have access to more nodes. This happens because nodes receive the finger table of the successor, so nodes from finger table are routed more often than the nodes that are not in the finger table. The joining procedure of AChord do a negative effect towards the anonymization, since it allow recording each node on the way to the successor. The SChord disable the gathering of information by active processes as well as AChord, Table 5.2 and 5.3. The limitation of the finger table exist too but is still more precise than those of the AChord version, Table 5.4 and 5.9. The join process in SChord remains recursive as in the origin Chord. To sum up the given evaluation results, the third example represents the global sight of the network via multiple spy-nodes.

5.3 Third example: Multiple Spies

For the third example, the properties are set as following: The simulation time is set to 180 minutes. The number of nodes increases each time beginning with 1015 nodes on first try and ending with 1111 nodes on the last. As the number increases the percentage amount for spy nodes increases too. The spy-nodes are represented by Australia, Germany and Italy.

Executed operations:
To accomplish a total overview, the whole functionality of all characteristics must be released on the simulation. Therefore, the node interact with each other, the none spying nodes as well as the spying nodes. The Latin American and the France groups execute some action like store and get. The Australian, German and Italian group perform the passive spy operation continuously after they join and the active spy periodically from the given point of time. The North American group do not act actively. This is the largest group of nodes in the given example, which have one simple purpose to remain undetected. The result of this simulation will show if it is possible, for the spy-nodes, to reconstruct the whole network with different types of members. The spy-nodes change the requested key each time they begin an active spy process. The change of the key for the active spy process aims to gather as much information as possible. The none spy-nodes request each times the same key. Because of the structure changes, when new group joins, the requests for the same keys will be routed via other nodes. The first group of spies join in the beginning of the simulation, the second in on the half period and the third near the end. The joining time has different effects to the behavior

Group name	Time	Operation
NorthAmerica	1m-30m	initApp false
Australia	15m-25m	initApp true
Australia	30m-45m	Spy
LatinAmerica	45m-55m	initApp false
Germany	60m-70m	initApp true
Germany	75m-80m	Spy
LatinAmerica	80m-90m	Store
France	100m-115m	initApp false
France	120m-130m	Get
Italy	135m-145m	initApp true
Italy	150m-160m	Spy
France	160m-170m	Get

Table 5.10: Example 3: Setup for the multple spy procedure.

to the results. When the first group join in, it maintains much more information from the passive spy process, because the Chord ring is reestablished during the simulation while new nodes are joining. The second group represent the average case, where the spies can see little changes reestablishing process. The last spy group join on the completed network, so only communication between the nodes are seen on the passive spy process.

Total number of nodes	1015	1024	1060	1111
Spy % of total number of nodes	0.29%	1%	4.5%	9%
Number of spy nodes	3	12	48	99
Chord known nodes	252/1015 (24.83%)	694/1024 (67.77%)	1024/1060 (96.6%)	1111/1111 (100.0%)
AChord known nodes	451/1015 (44.43%)	551/869 (63.41%)	710/880 (80.68%)	842/902 (93.35%)
SChord known nodes	56/1015 (5.52%)	349/1024 (34.08%)	690/1060 (65.09%)	942/1111 (84.79%)

Table 5.11: Example 3: Results of the multple spy procedure.

The results from Table 5.11, give a summary to the previously approached functionalities given in example one and two. The set-up of action on each try remains the identical, while the number of spy-nodes grows. According to this result, the AChord is the most vulnerable design for the anonymity manner. Foremost it is related to the iterative joining procedure described in the second example,

Table 5.8 and 5.9. Already at 0,29%, where only 3 nodes spies together, the known network size by those nodes is disastrous, because 3/1012 nodes knows almost 45% of the whole network. Table 5.11 the AChord's first column. Unfortunately, the tests with further spy nodes did not worked properly. This happened because some of spy nodes could not join the Chord ring in the given time. (This performance issue was mentioned in the last passage of the second example.) Still it is possible to glance at the not completed results. The origin Chord results are better, seems to be more anonymous, since they number of known nodes is lesser than in AChord. This is provided by the Table 5.11, note that on more simulation with other functionalities the result may appear different. Although the result of the Chord are better than those of AChord, it does not mean that the Chord version provide more anonymity, because the Chord spy can learn the network over time by executing the active spy operations, see therefore Table 5.2 and 5.3. Still with only 3 Chord nodes it is possible to reconstruct 24,8% nodes of the network and with 4,5% of spy-nodes, according to the network size, the Chord variant is able to cap almost the whole network 96,6%. The SChord is an improved version of the AChord design and is the best anonymous Chord variant evaluated in this paper, since it maintains the lowest amount of known nodes. Nevertheless, even here the impact of the multiple spy-nodes is huge, because only 1% of the network nodes are able to gain approximately 35% of information from the network scale.

Those examples represent only snap-shots for the existing solutions, with other set-up properties the evaluations results may be different. Further, the AChord solution is more anonymous than the origin Chord when the time and key for active spy process are set specific.

Chapter 6

Conclusion

This Chapter sums up the results of this bachelor thesis and gives a lead to the future work.
This bachelor thesis considers the anonymization of *Chord* [10] overlay. Therefore, an anonymity service SChord is designed on the existing *AChord* [12] solution. The two services SChord and AChord are implemented on the Chord overlay. Then AChord, SChord and Chord are evaluated and compared by different anonymity aspects via spy-nodes. Spy-nodes can act passively and actively. Thereby, the anonymous intercommunication between initiator and responder is tested through the active processes. The passive processes represent an eavesdropper who tries to find the initiator of the request (sender anonymity) and/or the responsible for the content (receiver anonymity). On the two implemented anonymity services AChord and SChord, the sender anonymity and the anonymous intercommunication are clearly tested and evaluated. Nevertheless, the receiver anonymity aspect can not be guaranteed completely, because of the structure of the Chord ring i.e. finger table.

Although Chord does not concern about any anonymity aspects, in some of the evaluated examples, the Chord node knows less information about the network than the AChord node. This is related to the design of the join process, while Chord has the recursive join the AChord does it iteratively. This means each node communicate directly with the joining node, this is where the new node collects a lot of information about the network. Additionally, the imprecise finger table increases the amount of next hop nodes on the routing path. Basically, the "later" (on greater the network size) an AChord node joins the network, the more information it collect form the network. This is the point where Chord may appear to be more anonymous, but the fact is that Chord is not anonymous at all. While an AChord node can only collect information about the network i.e. nodes on join and eavesdropping, Chord node can additionally record the responsible nodes form the active processes. So Chord does not provide anonymous intercommunication between the initiator and the responder. This anonymity condition on active process is possible for the AChord node, since the response is always returned by the own successor. With this main difference, the AChord node must be routed by each node of the network to record all network nodes. This is almost impossible because of the extremely limited finger table of AChord. While a Chord node, can learn a node by the key it requests. So even one

single Chord node is able to record i.e. reconstruct the whole network with enough keys and time. SChord is an improved version of the AChord design. The join process in SChord remains recursive as in the origin Chord to limit the knowledge of the network nodes for the joining node. Where the active routing process in SChord is exact the same as in AChord. The limiting conditions of finger table has been changed, so the finger table is more precise i.e. contains more different nodes that in the AChord design.

According to the given results, SChord is the best anonymous solution while it knows the minimal number of participants in the overlay, in comparison to the AChord and Chord solutions. The AChord solution provide more anonymity on the routing process than Chord but because of the disadvantages in the design of the join process it may collect, under certain circumstances, more information of the network than Chord. Still AChord provide anonymity services while Chord does not concern about the anonymity at all.

6.1 Future Work

Still the finger table impact for the anonymity must be considered further on greater number of nodes. This is necessary because the given examples were too small to evaluate the finger table properly. Further, the now designed and implemented model does not consider the leave procedure, the structure changes were simulated only via the joining of nodes, but the nodes did never leave the network.

Bibliography

[1] Michael K. Reiter, Aviel D. Rubin
Crowds: Anonymity for Web Transactions.
ACM Transactions on Information and System Security (TISSEC) 1.1 (1998): 66-92.

[2] Mesut Günes, Udo Sorges, Imed Bouazizi
ARA – The Ant –Colony Based Routing Algorithm for MANTEs.
Parallel Processing Workshops, 2002. Proceedings. International Conference on. IEEE, 2002.

[3] David Chaum
Untraceable Electronic Mail, Return Address, and Digital Pseudonyms.
Volume 24, Communications of the AMC, February 1981.

[4] *http://research.microsoft.com/enus/um/people/antr/Pastry/*

[5] Michael G. Reed, Paul F. Syverson, David M. Goldschlag
Anonymous Connections and Onion Routing.
Selected Areas in Communications, IEEE Journal on 16.4 (1998): 482-494.

[6] Aameek Singh, Bugra Gedik, Ling Liu
Agyaat: mutual anonymity over structured P2P networks.
Internet Research 16.2 (2006): 189-212.

[7] Ian Clarke, Oskar Sandberg, Brandon Wiley, Theodore W. Hong
Freenet: A Distributed Anonymous Information Storage and Retrieval System.
Springer Berlin Heidelberg, 2001.

[8] *https://sites.google.com/site/peerfactsimkom/*

[9] Charles W. O'Donnell, Vinod Vaikuntanathan
Information Leak in the Chord Lookup Protocol.
Fourth International Conference on. IEEE, 2004.

[10] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, Hari Balakrishnan
Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications.
ACM SIGCOMM Computer Communication Review, 31(4), 149-160.

[11] Antony Rowstron, Peter Druschel
Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems.
Microsoft Research Ltd and Rice University.
IFIP/ACM Middleware. 2001.

[12] Steven Hayel, Brandon Wiley
Achord: A Variant of the Chord Lookup Service for Use in Censorship Resistant Peer-to-Peer Publishing Systems.
Peer Publishing Systems, Proc. Second Int'l Conf. on Peer to Peer Computing, 2002.

[13] Giuseppe Ciaccio
Recipient Anonymity in a Structured Overlay.
Telecommunications, 2006. AICT-ICIW'06. International Conference on Internet and Web Applications and Services/Advanced International Conference on. IEEE, 2006.

[14] Giuseppe Ciaccio
Evaluating Sander and Recipient Anonymity in a Structured Overlay .
Privacy Enhancing Technologies. Springer Berlin Heidelberg, 2006.

[15] Leucio A. Cutillo, Refik Molva, Thorsten Strufe
Safebook: A Privacy-Preserving Online Social Network Leveraging on Real-Life Trust.
Communications Magazine, IEEE 47.12 (2009): 94-101.

[16] Andreas Binzenhöfer, Dirk Staehle, Robert Henjes
Estimating the size of a Chord ring.
University of Würzburg, Tech. Rep. 2004

[17] *http://www.graphviz.org/*

[18] *http://www.lix.polytechnique.fr/tomc/P2P/*

[19] *http://www.anonymousp2p.org/programs.html*

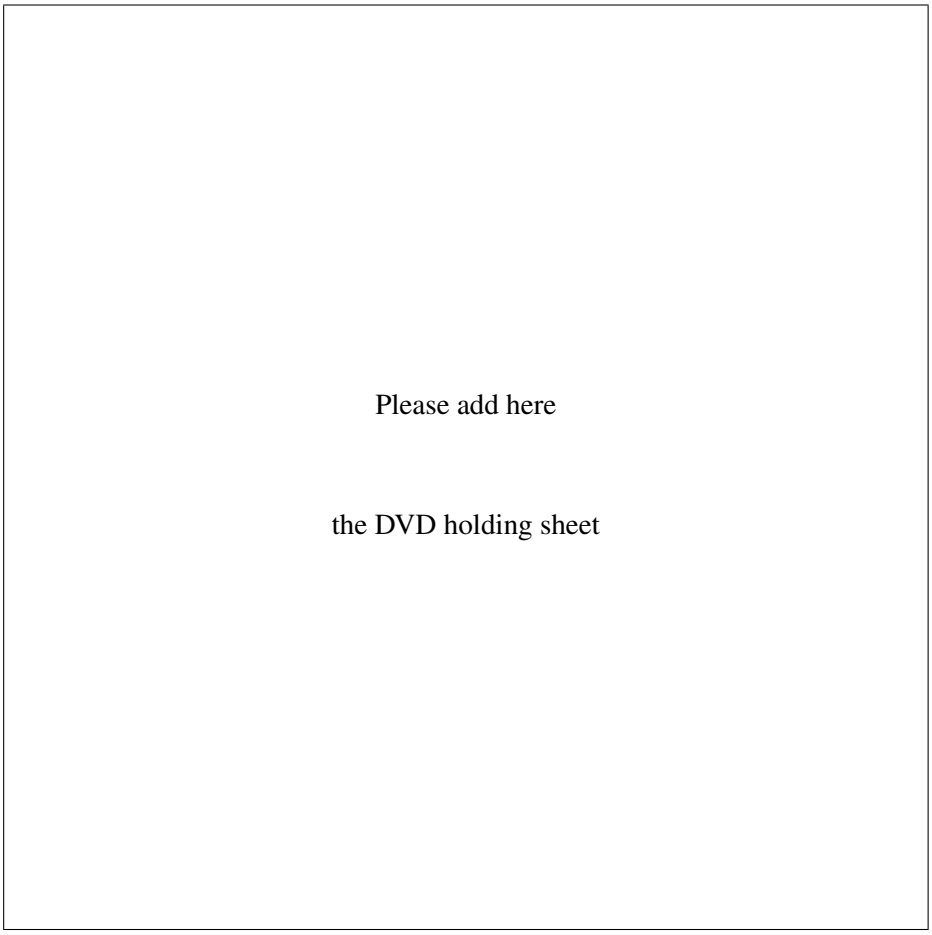
[20] Marco Dorigo, Vittorio Maniezzo, Alberto Colorni
Ant System: Optimization by a Colony of Cooperating Agents.

IEEE Transactions on Systems, MAN and Cybernetics-Part B: Cybernetics, Vol. 26 No.1, February 1996

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 17.Oktober 2014 Andrej Morlang



This DVD contains:

- A *pdf* Version of this bachelor thesis
- All LaTeXand grafic files that have been used, as well as the corresponding scripts
- **[adapt]** The source code of the software that was created during the bachelor thesis
- **[adapt]** The measurment data that was created during the evaluation
- The referenced websites and papers