UNIVERSITY OF MANNHEIM

Blacklisting Malicious Web Sites using a Secure Version of the DHT Protocol Kademlia

by

Philipp C. Heckel

A thesis submitted in partial fulfillment for the
degree of Bachelor of Science

in the
Faculty of Mathematics and Computer Science
Laboratory for Dependable Distributed Systems

May 2009

Declaration of Authorship

I, Philipp C. Heckel, declare that this thesis and the work presented in it are my own.

Furthermore, I declare that this thesis does not incorporate any material previously submitted for a degree or a diploma in any university; and that to the best of my knowledge it does not contain any materials previously published or written by another person except where due reference is made in the text.

Signed: ______________________________

Date: ______________________________

Contents

Declaration of Authorship **i**

List of Figures **iv**

List of Listings **v**

Abbreviations **vi**

1 Introduction **1**
1.1 Motivation 1
1.2 Scope 2
1.3 Road Map 3

2 Related Work **4**
2.1 Google Safe Browsing 4
2.2 Internet Explorer SmartScreen 5

3 Fundamentals **6**
3.1 Peer-to-Peer (P2P) 6
3.1.1 P2P Network Classification 7
3.1.2 Distributed Hash Tables (DHT) 8
3.1.3 Selected DHT Protocols 9
3.1.4 Kademlia 11
3.1.5 Potential Attack Scenarios 13
3.2 Public Key Infrastructures (PKI) 15
3.2.1 Types of Cryptography 15
3.2.2 Motivation 16
3.2.3 Public Key Cryptography 17
3.2.4 Entities in a Public Key Infrastructure 18
3.2.4.1 Certificates 18
3.2.4.2 Certificate Authorities (CA) 19
3.2.5 Standard Specifications 20
3.2.5.1 X.509 Certificates 20
3.2.5.2 Transport Layer Security (TLS) 21

4 Application Design **24**
4.1 Initial Situation and Goals 24
4.1.1 Blacklist Service 24

4.1.2 P2P-based Core System 26
4.2 Design Discussion 28
4.2.1 Peer-to-Peer as Underlying Technology 29
4.2.2 Kademlia as DHT Protocol 29
4.2.3 PKI for Peer-to-Peer 30
4.2.4 Secure Extension for Kademlia 32
4.3 Basic Architecture 33
4.3.1 Overview 33
4.3.2 KadS: Core Network 34
4.3.3 Blacklist Service 37

5 Implementation **40**
5.1 Core Network: The Secure Peer-to-Peer Protocol 40
5.1.1 Terminology 40
5.1.2 Core Interface `DistributedHashMap` 42
5.1.3 Local `KadS` Node 42
5.1.4 Initial Authentication: `HandshakeClient` and `HandshakeServer` . 43
5.1.5 Messaging System: `Messenger` (UDP) 46
5.1.6 Kademlia RPCs: `Operations` 48
5.1.7 Secure Communication and Encryption 49
5.2 URL Blacklist 50
5.2.1 Blacklist Node 50
5.2.2 Client Classification 51
5.2.3 Protocol: `BlacklistServer` and `QueryServer` 52
5.3 Examples 54
5.3.1 KadS Node 54
5.3.2 Blacklist Node and Clients 55

6 Testing **58**
6.1 Setup 58
6.2 Average RTT of KadS' `put`-Operation 59
6.3 Static Load Test on a Blacklist Node 61

7 Conclusion **63**

Bibliography **66**

List of Figures

3.1 Hybrid and Pure P2P Networks 8
3.2 Kademlia's FIND_NODE Operation 13
3.3 PKI: Hierarchic and Flat Trust Models 19
3.4 Self-signed Root CA 21
3.5 TLS Handshake 23

4.1 Schematic Design of a Blacklist Node 34
4.2 Schematic Design of a KadS Node 35

5.1 Schematic Process of a Browser Client 41
5.2 KadS Handshake 44
5.3 KadS Message Process: Lifecycle of a UDP Packet 48
5.4 Blacklist Protocol & Handshake 52

6.1 Average RTT of KadS Operations 59
6.2 Static Load Test on a Blacklist Node 61

List of Listings

5.1 Core Interface DistributedHashMap . . . 42
5.2 Concurrency via Thread Pools . . . 45
5.3 HandshakeFinishedListener Interface . . . 45
5.4 Implementation of Kademlia's STORE-Operation . . . 47
5.5 Example: A KadS Config File . . . 54
5.6 Example: Using a KadS Node . . . 55
5.7 Example: Using the TrustworthyClient class . . . 55
5.8 Example: Using the UntrustworthyClient class . . . 56
6.1 Test the average RTT for the put-Operation . . . 59

Abbreviations

ACK	Acknowledgement
AES	Advanced Encryption Standard
API	Application Programming Interface
CA	Certificate Authority *or* Certification Authority
CN	Common Name
DES	Data Encryption Standard
DSA	Digital Signature Algorithm
DHT	Distributed Hash Table
DN	Distinguished Name
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IMAP	Internet Message Access Protocol
IP	Internet Protocol
KadS	Kademlia Secure
NNTP	Network News Transfer Protocol
NOP	No Operation
P2P	Peer-to-Peer
PKI	Public-Key Infrastructure
POP	Post Office Protocol
RPC	Remote Procedure Call
RTT	Round-Trip Time
RSA	Algorithm designed by Rivest/Shamir/Adleman
SHA	Secure Hash Algorithm
SK	Secret Key *or* Session Key

SMTP	Simple Mail Transfer Protocol
SSL	Secure Socket Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TTP	Trusted Third Party
UDP	User Datagram Protocol
URL	Uniform Resource Locator
XOR	Exclusive OR

Chapter 1

Introduction

As the Internet has evolved to the powerful network it is today, more and more criminal minds try to exploit it for their needs. Internet crime has become a dangerous threat to both home users and companies. According to the Internet Crime Complaint Center, the amount of complaints linked to Internet fraud hit a new record in 2008 by causing a total loss of $265 million. The fact that this number almost quadrupled in only four years demonstrates that cyber crime rates are rising and the need for protection against it is higher than ever [1].

1.1 Motivation

Not only has the amount of crime on the Web risen over the years, but also the types of attacks have changed significantly. While phishing emails and malicious attachments were the major infection vectors in the past, so called *drive by*-downloads on malicious Web sites now form the overwhelming majority of Web-based attacks [2]. That is, Internet users' workstations get infected with malicious software (malware) without their knowledge by simply browsing a compromised Web site. The malware installed on the user's workstation is mostly designed to either steal information such as bank account data or passwords, or can be used by the attacker to control a botnet. Especially in 2007-2008, more trojan programs were developed and distributed via Web sites than ever before. In fact, the virus analysts of Kaspersky Lab believe that the number of malicious Web sites and malware programs this year will even exceed the one from 2008 [3].

Given these facts, it is crucial to protect the users' workstations from being infected. Many organizations developed software and invented defense techniques against those

attacks. However, most solutions such as anti-virus protection or software based firewalls are rather reactive and leave security updates to the user.

1.2 Scope

The approach suggested in this thesis is to *prevent* users from browsing malicious Web sites by providing a service to check a Web site for malignity *before* the user opens it. Hence if a Web site has been reported to be malicious, the browser can warn the user and suggest not visiting it.

To provide a system like this, three basic components are required:

- Core System: A publicly reachable database providing a service to query it for the malignity of URLs or domains.
- Honeynet Client: A trustworthy instance that identifies malicious Web sites and fills the database with up-to-date information.
- Browser Plug-In: A client for querying the database for the malignity of an URL or domain, possibly in form of a browser plug-in.

Two components of this system already exist or are being developed by other instances, respectively. The *honeynet clients*[1] are already up and running and collect information about malicious Web sites. The *browser plug-in* is currently being developed in the scope of another thesis. Thus, this thesis' main objective is to design and implement the *system core*, i.e. a database to store information about Web sites. This also includes an interface for both clients to enable interaction with the system.

In contrast to the obvious solution to realize the service on a classic client-server basis, the proposed system design uses a secure *distributed hash table (DHT)* to reduce the load of single systems and to be more resistant against denial-of-service attacks, or general failures. Furthermore, it addresses the major disadvantage of most distributed systems by taking security into consideration. In fact, it combines the flexibility of a *peer-to-peer (P2P)* based system with the security of a *public key infrastructure (PKI)* by creating a PKI-supported distributed hash table, called *Kademlia Secure (KadS)*.

[1] Namely the honeynet client cluster of the Laboratory for Dependable Distributed Systems of the University of Mannheim

1.3 Road Map

The thesis splits in five main chapters.

In order to present a wider view of the discussed topic, chapter 2 introduces two well-known similar solutions to the above described problem: Google Safe Browsing and Microsoft SmartScreen were developed to address the problem of malicious Web sites, but are based on a different core system.

Chapter 3 presents required fundamentals and principles, and explains the two basic concepts used for this thesis' application. The first part explains the concepts of the *peer-to-peer paradigm*, introduces distributed hash tables and the protocol used for this thesis (Kademlia). The second part concentrates on giving an idea of what *public key infrastructures* are about and explains the fundamental concepts of cryptography.

After focusing on the used technologies, chapter 4 defines the goals of the application, discusses various design approaches, and introduces the final application design. While the first and second part of the chapter concentrates on developing the most suitable approach, the last part illustrates the derived architecture of the secure DHT (KadS) and the URL blacklisting service.

Chapter 5 finally presents the results of the design process by introducing the concrete Java-based implementation. It discusses how the design specification was realized and which parts had to be added to or altered in the Kademlia protocol to create a secure P2P network. The first part of the chapter introduces the KadS protocol design and its specific Java implementation. The second part covers details about the URL blacklisting service, the application for which the KadS protocol was designed.

In order to briefly evaluate the quality of the developed implementation, chapter 6 focuses on analyzing the road capability of the system. By performing several tests at different network sizes, the application has to demonstrate its performance and withstand static load tests.

Chapter 2

Related Work

The main purpose of the system is to identify malicious Web sites via a remote URL blacklist. The end-user clients in this scenario are common Web-browsers such as Firefox, Safari or the Internet Explorer. Since the basic idea is not new, few big companies already designed and implemented similar services.

This chapter briefly introduces two existing solutions: Google's *Safe Browsing* for Firefox/Safari and Microsoft's *SmartScreen* for the Internet Explorer.

2.1 Google Safe Browsing

Initially designed and developed by Google and distributed as part of the Google Toolbar, the former *Safe Browsing* extension [4] is now licensed under the Mozilla Public License and an essential part of Firefox and Safari [5, 6]. The component checks the sites a user visits against regularly downloaded lists of reported phishing and malware sites. If a URL or domain matches an entry in the list, a warning message is displayed to the user. In the newest version, it also supports live lookups with up-to-the-minute fresh lists for every URL instead of using the cached local versions.

Since the protocol is well-structured and openly defined, the provided lists could come from any server that implements the system. However, due to its origin, both browsers use the Google servers by default. That is, the lists of phishing and malicious Web sites are maintained by Google which, according to ZDNet, uses a "*combination of automatic (honeyclients) and community-driven efforts to analyze a Web site*" [7].

The protocol is based on simple HTTP request/response-cycles and supports blacklists as well as white-lists. It differentiates between malware-, phishing- and white-lists and

supports various list formats, including regular expression lists or hashed lists of URLs or domains, respectively. It typically updates them every 30 minutes and usually only compares the visited URLs to the local lists. However, if a Web site matches a local list entry, it double-checks the URL using a live lookup to make sure that the entry is still up-to-date [8, 9].

Surprisingly, Google's competitor Apple also uses the technology in its proprietary browser Safari. Apple silently included Safe Browsing in version 3.2 and only mentions it in its License Agreement [6, 10].

2.2 Internet Explorer SmartScreen

Since version 7 of the Internet Explorer, Microsoft also included phishing protection in its browser. The so called *Phishing Filter* detects only phishing attempts, but does not protect users from drive-by downloads on malicious Web sites [11]. The recently released Internet Explorer 8 extends the Phishing Filter by the missing anti-malware protection and has been rebranded to *SmartScreen* [12].

Like its predecessor and in contrast to Google's Safe Browsing API, SmartScreen mostly relies on live lookups to determine if a Web site has been reported to be a phishing site or distributes malware. Although it also keeps a regularly downloaded list of known safe sites, it queries Microsoft's server for most of the visited Web sites. That is, SmartScreen and Phishing Filter only check "*sites that aren't in IE's downloaded 'known-safe' list*" [11] and hence are able to use up-to-date information for most of the Web sites.

In addition to the blacklist approach, SmartScreen also statically analyzes each visited Web site for characteristics associated with known phishing attempts and warns if sites are suspicious.

Chapter 3

Fundamentals

This chapter explains required fundamentals and principles used in the implementation, and tries to illustrate the basic concepts. It is organized in two sections.

The first part introduces the peer-to-peer paradigm and compares it to the classic client-server model. Furthermore, it discusses distributed hash tables and presents the most important protocols. The second section concentrates on the two different concepts of cryptography and discusses their possible application domains. Moreover, it introduces the public key infrastructure and explains the most important standard specifications.

3.1 Peer-to-Peer (P2P)

With the growth of the World Wide Web over the last years, the requirements of Internet applications have changed significantly. While most traffic was originated from classic client-server applications in the early days of the Internet, nowadays 50-70% of the bandwidth is attributed to so called *Peer-to-Peer (P2P)* applications [13, pg. 370].

Oram et al. define a peer-to-peer system as "*a self-organizing system of equal, autonomous entities (peers)* [which] *aims for the shared usages of distributed resources in a networked environment avoiding central services*" [14].

As opposed to the client-server paradigm, in which the roles of participating entities are strictly distinct, the peer-to-peer model defines all participating peers as equal. That means, they are "client" and "server" at the same time and no central instance is able to control other computers, i.e. most of them are completely decentralized and self-organized. This fact entirely eliminates many problems in terms of attack resistance, scalability and flexibility, which can be typically found in client-server environments:

- No single point of failure/attack: Due to the lack of a central server, it is more difficult for attackers to disrupt the service provided by the P2P network. Most P2P systems are designed to be redundant and the failure of few peers does not affect the service quality. In fact, P2P services mostly are more reliable and fault tolerant than client-server systems [13].

- No resource bottleneck: In client-server based systems, a lack of resources such as processor time or memory shortage is more likely to occur. P2P networks distribute resources of interest equally amongst the participating peers and each node uses resources of the others.

- Scalability and flexibility: In order to provide a flexible environment, P2P networks allow peers to join and leave the network as they like. Hence, if the network reaches a peak in terms of resource usage, one can simply add new peers to scale the application and balance the load among all peers.

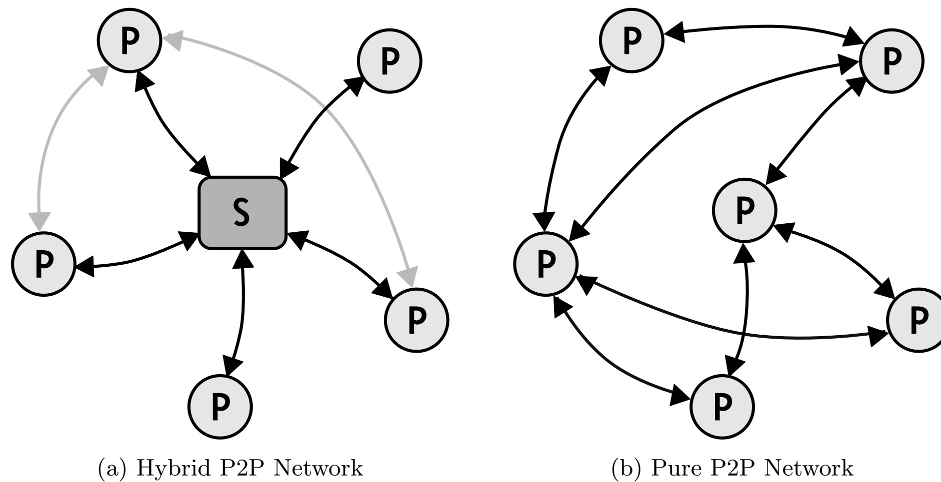
These facts point out a few characteristics of peer-to-peer and demonstrate the power of its architecture. Steinmetz/Wehrle even believe that the client-server model "*can no longer fully meet the evolving requirements of the Internet*" [13, pg. 9].

However, client-server systems are still popular and an essential part of the Web's infrastructure. Most everyday applications and protocols, e.g. HTTP, POP or IMAP, use this centralized approach rather than P2P. The technology is more mature and traditional servers are relatively easy to develop. Also, not every application is practicable to work within a decentralized system. This effectively leaves it to IT Executives and Software Architects to evaluate the suitability of an application to the P2P model.

3.1.1 P2P Network Classification

Literature typically classifies peer-to-peer networks according to the amount of central services involved as well as their internal structuredness.

Networks that rely on any kind of centralized service, e.g. a name resolver or a resource location database, are commonly called *hybrid* P2P networks (fig. 3.1a). They generally use a central service to locate other nodes or resources in the network. Typical examples include file-sharing networks such as the original Napster in which the centralized supernode keeps track of which files are held by which peers, but the actual transfer happens without any interaction of the central server. Opposed to that, *pure* P2P networks such as Gnutella or Freenet only consist of equal peers and no super-peers are involved (fig. 3.1b). According to Yang/Garcia-Molia, hybrid networks "*have better performance than*



(a) Hybrid P2P Network (b) Pure P2P Network

FIGURE 3.1: Hybrid P2P networks use a central server to locate resources in the network, but connect other peers to exchange information. Pure P2P networks consist of equal peers only.

pure systems because some tasks (like searching) can be done much more efficiently in a centralized manner" [15]. Even though this is true for most environments, it does not take scalability issues into consideration. Especially in extremely big networks, the central service might not be able to handle all requests in time and hence eliminate the hybrid-bonus.

Another way of classifying P2P networks is to regard the structuredness and degree of decentralized decision making within the system. *Unstructured* models usually do not set any constraints and hardly regulate the network in terms of resource distribution or growth policies. The placement of content is not related to the topology of the system and peers are allowed to connect to one another as they like. Therefore, over time, the network forms a random graph of connected nodes without any systematic distribution. The major disadvantage of this approach is that due to the lack of structure, each search query causes a flood of messages to many peers and hence is very bandwidth intensive.

Structured models solve this problem by enforcing strict constraints on resource placement and network evolution. Models are typically designed to support a fast search algorithm with sub-linear complexity. That makes it possible to locate any resource in the network with only few hops [16].

3.1.2 Distributed Hash Tables (DHT)

Structured peer-to-peer systems mostly focus on providing a "*distributed, content-addressable data storage*". Instead of identifying resources via their network location, the system is designed to store the content itself at a specific position in the network. These

so called *Distributed Hash Tables (DHT)* offer significant advantages. Not only are they more fault-tolerant and reliable than unstructured approaches, they also outperform them in terms of scalability and performance. Especially the latter differentiate the system from first-generation P2P networks. Since most operations of common DHT protocols have a complexity of $O(log\ N)$ or $O(log^2\ N)$, adding many peers to the network hardly changes the performance at all [13, pg. 9-16].

The first DHT protocols, i.e. Chord [17], Pastry [18], CAN [19], and Tapestry [20], were designed in 2001 when the research community realized their enormous potential. They all differ in data management and routing strategies, but essentially follow the general paradigm of *consistant hashing* [21]: In contrast to classic hash tables in which changing the number of array slots results in the recalculation of all hash-keys, consistant hashing allows resizing the table without having to change the keys. It was originally designed to solve the problem of a varying number of machines in a network and is now used by DHT protocols. The idea is to assign each node a k-bit identifier and divide the address space, typically $\{0, 1\}^k$ for $k > 0$, in roughly equally sized segments (or buckets). Each node is assigned to a segment and is "*responsible for storing all the data items with hash values that fall within its assigned segment*" [16]. This fixed structure makes it possible for peers to locate the responsible node(s) for a given key, and thus store or retrieve data items.

3.1.3 Selected DHT Protocols

Having the basic structure in common, DHT protocols differentiate most in terms of lookup procedures and how connections between nodes are organized. The following sections examine four of the most important and interesting DHT protocols and point out their differences.

Chord

Chord [17] was developed at the Massachusetts Institute of Technology (MIT) in 2001 and is considered one of the original distributed lookup protocols. The only operation it supports is mapping a key onto a node in the network. That means, Chord is not a full DHT protocol, but only allows locating the responsible node for a given key. The distributed *(key, value)*-table, can be implemented on top of it.

Chord implements a distributed routing table, that is, it needs only routing information about a few other nodes. It is organized in a *ring* architecture in which each peer knows a few of its successors and its predecessors. Search queries for a given key are passed around until the responsible node is found. In order to not having to walk through the

whole ring, each Chord node additionally keeps a relatively small *finger table*. The table stores succeeding nodes in a distance of 2^i, which allows exponentially growing jumps through the ring. As Chord only provides a *key* $\rightarrow$ *node* mapping and not a full DHT, the operation ends as soon as a responsible node is found [22].

CAN

Content Addressable Network (CAN) is "*a distributed infrastructure that provides hash table-like functionality on Internet-like scales*" [19]. It was designed at the University of California in 2001 and is based on a multi-dimensional Cartesian coordinate-system. Using the consistant hashing approach described above, it divides the identifier address space in d-dimensional segments and assigns nodes to be responsible for a particular space range, called *zone*. That is, CAN effectively maps keys of data items to a point in the network's coordinate system. Each node keeps a list of the nodes responsible for neighbor-zones so that queries can be routed through different zones [22].

CAN's performance strongly depends on the number of nodes and dimensions in the network. The more dimensions the network consists of, the more neighbors have to be administered by each node. Unlike most DHTs, a lookup query in a CAN has a complexity of $O(n^{1/d})$, i.e. it is not logarithmic. However, for networks with up to 30 million nodes and 6 dimensions, for instance, the routing path length is even shorter than for DHTs with a logarithmic approach. That means, fewer nodes have to be queried to find the responsible peer [19].

Pastry

Pastry is "*a generic peer-to-peer object location and routing scheme, based on a self-organizing overlay network of nodes connected to the Internet*". Designed at the Rice University in cooperation with Microsoft, its main focus lies on being fault-tolerant, scalable and reliable.

Similar to Chord, Pastry has a k-bit circular identifier space and node IDs are assigned randomly. Its default routing algorithm is based on the numerical closeness between identifiers. Like Tapestry and Kademlia, which will be discussed in later sections, messages are routed to the node that shares the longest prefix with the given key. The lookup operation ensures that the shared prefix-length increases with every iteration until the responsible node is reached.

To achieve a better performance, Pastry "*seeks to minimize the distance messages travel*" by providing an adjustable network locality. In contrast to other DHT protocols, Pastry does not dictate a proximity metric but allows the application to provide it. That is, the distance between nodes could, for instance, be a value based on traceroute-results or

the number of IP routing hops in the Internet [18]. This allows a flexible adjustment of the network environment according to the applications requirements, e.g. performance or reliability.

Tapestry

Like CAN, Tapestry was developed at the University of California in 2001. The original paper describes the DHT protocol as "*an overlay location and routing infrastructure that provides location-independent routing of messages directly to the closest copy of an object or service using only point-to-point links and without centralized resources*" [20].

It is based on the *Plaxton mesh* [23], a distributed data structure for locating and routing messages across an arbitrarily-sized network. Due to the fact that the original Plaxton approach assumes a constant population of nodes, Tapestry extends its design to support P2P-desired properties such as fault-tolerance, redundancy and caches. It offers a greater "*system-wide stability by bypassing failed routes and nodes*" and is able to reorganize the network's topology on the fly.

Similar to Pastry and the later described Kademlia, Tapestry forwards messages according to the longest shared prefix of two identifiers. To support fast routing, it keeps relatively small local *neighbor maps* which contain pointers to the closest nodes of each prefix [22].

3.1.4 Kademlia

Kademlia, a DHT protocol designed by two students of the New York University in 2002, provides "*provable consistency and performance in a fault-prone environment*" [24]. The protocol is very similar to Pastry [18], but improves it in terms of its routing algorithm, and adds concurrency parameters to the lookup routine. Due to its fault-tolerant design and its good performance, it has been implemented in various applications such as Overnet, eMule or Azureus DHT, and serves as the underlying core system of this thesis' service.

Like many other DHT protocols, Kademlia requires every peer that wants to participate in the network to create a 160-bit *identifier* (ID) and the keys of the DHT to be of the same length. This identifier serves as unique key to identify nodes and DHT entries on the one hand. More importantly, it allows the calculation of the distance between them on the other hand. That is, the identifier defines the logical position in the network and makes it possible to locate the *closest* nodes to a given ID.

XOR Metric

Closeness within the network is defined by the *XOR metric* and has nothing to do with the actual physical closeness of nodes or DHT entries. The distance between identifiers is the result of the bit-wise *exclusive OR*-function (XOR) and therefore treats them as leaves in a binary tree [13]. The distance between two identifiers $id_1 = 10111\ldots$ and $id_2 = 10001\ldots$, for instance, is calculated by simply *XOR*ing them:

$$
\begin{aligned}
d(id_1, id_2) &= id_1 \oplus id_2 \\
&= 00110\ldots
\end{aligned}
$$

Concurrency

In contrast to other DHT protocols, Kademlia's routing algorithm uses parallel RPCs to speed up the lookup performance: Instead of sending one message at a time and waiting for its result, Kademlia introduces the concurrency parameter α that allows to trade network bandwidth for a better latency and fault recovery [22]. Kademlia always queries α nodes concurrently and waits for their responses. As soon as the first response arrives, it queries the next node so that always α queries are on their way.

The authors of the protocol suggest a value of $\alpha = 3$, but allow programmers and administrators to adjust it as they like. The higher the value, the more UDP packets are exchanged during a lookup, but the faster is the RPC.

Remote Procedure Calls

Kademlia is based on four simple remote procedure calls (RPCs): PING, STORE, FIND_NODE and FIND_VALUE. While PING simply probes whether a node is alive and STORE instructs a remote node to store an entry in its locale hash table, the most important operations of the Kademlia protocol are the FIND_*-RPCs. FIND_NODE locates the K closest nodes to a given identifier in the network by iteratively asking nodes of the local routing tree if they know any closer nodes. If the queried nodes respond with so far unknown but closer nodes, the initiating node queries them as well until no closer nodes are found. Figure 3.2 illustrates this process.

The FIND_NODE-operation is particularly important for entry lookups, but also when a node wants to store a new entry in the DHT. Kademlia stores entries of the distributed hash table at the K closest nodes to the entry key (while K typically equals 20). To store a new entry in the DHT, the local node must first locate the K closest nodes and then send a simple STORE-RPC to each of them. That way, each value is stored at K nodes and even if some of them fail or go offline, the value is still available in the network.

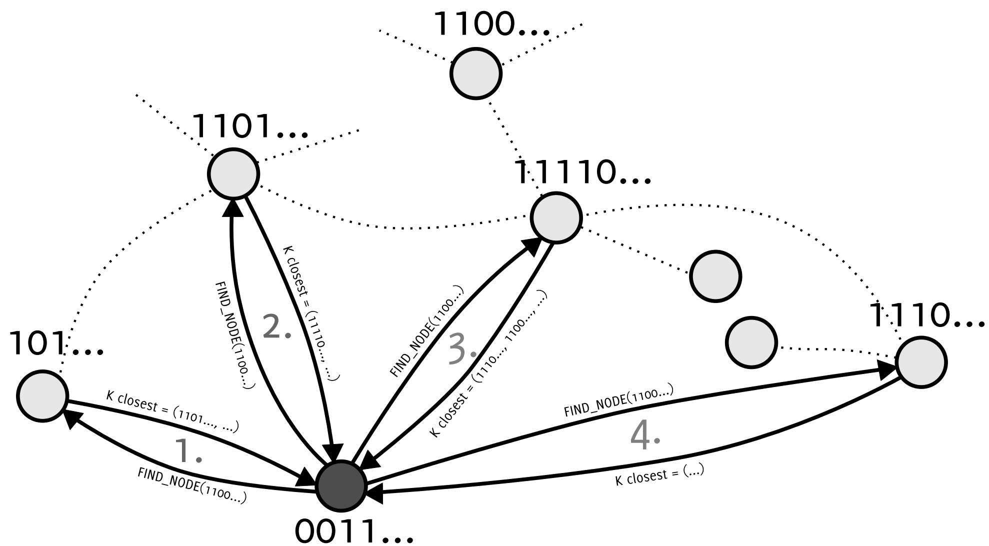


FIGURE 3.2: Kademlia's FIND_NODE-RPC locates the K closest nodes of the network relative to a given identifier: The initiating node asks the closest nodes of the local routing tree to return their list of the K closest nodes, and iteratively also asks the returned nodes. Each iteration, it asks a closer node until no unasked closer nodes are returned. In this example, the node 0011... tries to locate the identifier 1100... by querying the known node 101... Subsequent RPCs are to nodes returned by previous RPCs.

The FIND_VALUE-operation behaves almost identical as the FIND_NODE-operation. The only difference is that if a node receives a FIND_VALUE-RPC and knows the value for the queried key, it returns the value instead of the K closest nodes. As soon as the initiating node receives the value from any node, it returns the value and terminates the operation.

3.1.5 Potential Attack Scenarios

Even though the above introduced DHTs differ in terms of routing algorithms or distance metric, they implement a very similar concept. In fact, all of them share the idea of a distinct *node identity* and depend on evenly distributed node IDs.

A well-known attack scenario, the *Sybil attack*, uses this property to disrupt the network: In distributed hash tables, the node ID is not only an identifier of the peer, but also specifies its position within the network and sometimes even the closeness to DHT entries. This supports fast lookup algorithms on the one hand, but also makes it possible for an attacker to instantiate numerous nodes with arbitrary node IDs on the other hand. That is, by flooding the network with well-considered nodes, an attacker "*can control a substantial fraction of the system*" [25].

The attacker could, for instance, remove items from the network by placing fake-nodes around the target key. Instead of storing and responding properly, the nodes only pretend to store items when asked to, and never return any entries. This would effectively make the real table entry invisible to other nodes, and hence unreachable.

Another possibility is the strategic positioning of nodes within the network. Depending on the DHT protocol, an attacker could disrupt the routing by placing fake-nodes around and in a certain address space. By returning false routing information, the attacker could prevent nodes from outside the target area to reach the nodes inside. That is, it is possible to cut off whole parts of the network.

3.2 Public Key Infrastructures (PKI)

As long as people have communicated with each other, there has been a desire to secretly exchange information without the knowledge of others – be it a business competitor or an opponent in war. Famous examples of cryptography in history include the *Caesar cipher*, a simple encryption technique used during military campaigns by Julius Caesar, or the *Enigma*, an electro-mechanical machine used for encryption and decryption during World War II. No matter the purpose, the importance of secure data exchange today is higher than ever.

The Internet, originally designed for a smaller scale and hence not equipped with cryptographic functions by design, complicates secret exchange of information even more. Most of the everyday application protocols such as HTTP or SMTP originally have been designed as plain-text protocols without any form of encryption. Only in recent years, the *Internet Engineering Task Force* and other organizations developed secure pendants for most of them, or the *Secure Socket Layer (SSL)* is used to encrypt the connection, respectively.

In any case, the *public key infrastructure (PKI)* technology plays an important role to realize secure data exchange solutions and enables modern applications such as global shopping or online banking.

3.2.1 Types of Cryptography

The main goal of cryptography is to transform a plaintext into ciphertext ("gibberish") and back, in a way that an entity without a matching key is not able to extract the original message. Currently, two categories for performing these tasks exist:

Secret-key cryptography uses a mathematical encryption function with a secret key as input value to transform a plaintext into a ciphertext; and a reverse function using the same key to decrypt it again. That is, only *one* key is used for both encryption and decryption which is why this method is also referred to as *symmetric* cryptography. The whole mechanism is called a *cipher*. Well-known examples for symmetric algorithms include the very simple ROT13, in which "*each letter in the original message is replaced with the letter 13 places ahead*" [26, pg. 8], as well as more secure and modern ones such as the *Data Encryption Standard (DES)* or the *Advanced Encryption Standard (AES)*.

Opposed to that, *public key* cryptography is an *asymmetric* approach that uses *two* secrets – one key for encryption and a different one for decryption. The two keys are mathematically related but cannot be used to calculate each other. Thus, one key can be

publicized and used by others to encrypt messages (*public key*), and the other key is kept secret and is used to decrypt messages (*private key*).[1] This approach was first proposed in the paper "New Directions in Cryptography" by the two researchers Witfield Diffie and Martin Hellman in 1976 [27] when they realized the partial fussiness of symmetric solutions. In fact, today's PKIs are entirely based on their idea which is why they are commonly considered the pioneers of public key infrastructure.

3.2.2 Motivation

Before getting into details about PKI, it is useful to understand what benefits the technology brings and why the effort of implementing it is worth it.

Even though symmetric cryptography offers desirable characteristics such as very fast and small implementations in hard- and software, it raises questions in terms of scalability, manageability, etc.

Adams/Lloyd [26] define three main reasons why the asymmetric approach is more suitable for many applications:

- The need for secret key exchange: The concept of symmetric encryption relies on the fact that both communication partners share a secret key. Only if they know the key, they are able to encrypt messages before sending, or decrypt them after receiving, respectively.
- Difficulties in initiating a secure connection between unknown parties: In order to create a secure connection, it is hence necessary to negotiate a key over insecure, plaintext channels. This additional step complicates the ease of symmetric encryption and can be extremely difficult to implement. The first key exchange protocol that made this possible over an insecure communications channel was introduced by Diffie/Hellman in 1976 [27].
- Difficulties of scale: If more than two parties need to communicate, each pair of users needs a distinct secret key for their communication. If A and B negotiated a key K_{AB}, for instance, A still needs a different key K_{AC}, K_{AD},... for the communication with C, D,... to make sure that the confidentiality is not compromised. This would mean, that each user has to maintain up to $n^2/2$ keys what makes this approach impractical even for small user counts. Even though *Key Distribution Centers* solve this problem by introducing users to eachother, this requires a central instance to coordinate the communication.

[1] Besides encryption and decryption, public and private key can also be used for digital signatures and other services. Details will be discussed in later sections.

3.2.3 Public Key Cryptography

Asymmetric encryption solves these problems by the simple fact that one key can be publicized. This eliminates the need for a key exchange and enables encryption between to unknown parties. If an entity A has never communicated with B before, it can simply lookup the B's public key K_B^{pub} (e.g. in a public repository such as a key server) and use it to encrypt messages for B. B can then use its private key K_B^{priv} to decrypt the message. No secret keys have to be exchanged and even strangers can communicate securely.[2]

These two keys, i.e. the public key with its related private key, are commonly called a *key pair*. Both keys are completely different, but they are also related. Their relationship is based on the fact that the inverse functions of some mathematical operations have a considerably higher complexity. Multiplying two large integers, for instance, is a very simple and fast operation whereas the factorization is computationally infeasible even for super computers. Well-known algorithms taking advantage of this are, e.g., the algorithm proposed by *Rivest/Shamir/Adleman (RSA)* and the *Digital Signature Algorithm (DSA)*.

Public key cryptography uses this enormous inequality of computation time for a variety of services. Besides the encryption of data and secure communication between strangers, it also enables other services such as digital signatures and provable data integrity [26]:

- Digital signatures: Messages sent over an insecure channel can typically not be trusted because they are routed via various untrustworthy machines. Asymmetric cryptography allows the creation of digital signatures that prove the *data integrity* on the one hand, and the *message origin* on the other hand. By signing the original data with the private key and attaching the signature to the message, the recipient can verify its accuracy with the corresponding public key.
- Encryption: Not all known algorithms allow encrypting data; DSA, for instance, has been designed to support sign/verify-operations only and can hence not be used for encryption. RSA on the other hand can be used for both encryption/decryption as well as for signing/verification.
- Key agreements: Public key cryptography allows completely asymmetric communication so that theoretically no symmetric keys are required. However, in practice, only the initial *handshake* of two parties is asymmetric. The actual data transfer is mostly encrypted symmetrically because public/private-key operations are

[2] Note that this does not take trust issues into consideration. Details about signatures and verification will be discussed in later sections.

significantly slower than symmetric solutions. Therefore, a symmetric secret key is transferred or commonly generated in the handshake phase using asymmetric cryptography; And future data transfer is encrypted using the exchanged secret key.

3.2.4 Entities in a Public Key Infrastructure

Even though public key cryptography extends the functionality of the symmetric approach and supports it with key agreement techniques, it still does not provide a standardized application-usable infrastructure and lacks of a uniform trust-system. The *Public Key Infrastructure (PKI)* has been designed to solve these issues. A PKI is a self-contained and uniform infrastructure that provides *authentication*, *integrity* and *confidentiality* using public key concepts and techniques [26]:

To realize a system like this, PKI introduces a number of concepts and entities. The following paragraphs define the ones used for this thesis briefly.[3]

3.2.4.1 Certificates

The idea of using two different keys for encryption and decryption makes it possible to publicize the public key and enables secure communication. However, this approach presumes that the *correct* public key is retrieved on the one hand, and that it has not been *altered* by an attacker on the other hand. That is, public key cryptography only enables secure communication, but is not able to verify the *identity* of the communication participants. If, for instance, a user A wants to communicate with another user B who sent his public key K_B^{pub} attached to an e-mail, A neither can prove that the message was really sent by B, nor that the received public key is the correct one.

Public key certificates, or simply *certificates*, address this problem by adding provable identity information to the public key. A certificate can be defined as "*a data structure that contains some representation of the identity and a corresponding public key*" [26]. That is, it binds an identity such as the name and address of an individual to a given key pair.

To prove that the attached information matches the identity, PKI provides the possibility to sign certificates using the digital signature of another, *trusted* identity, also called a *Trusted Third Party (TTP)*. This way, the signing entity guarantees the authenticity of the signed certificate and passes on its trust.

[3]The concept of PKI is too extensive to discuss it in detail. A complete description can be found in "Understanding PKI", *Carlisle Adams and Steve Lloyd*, 2nd edition, 2007, chapter 3 [26]

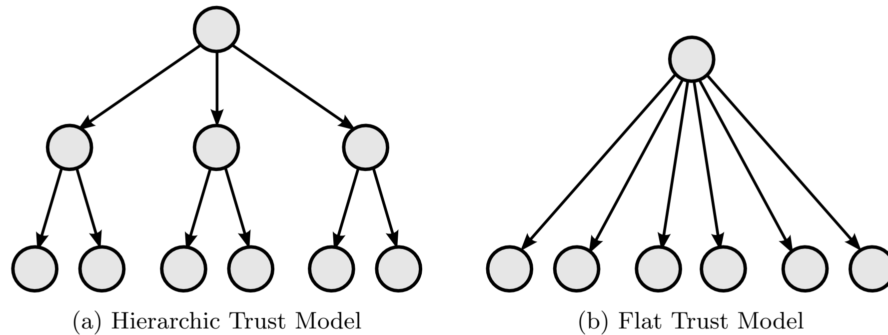


FIGURE 3.3: (a) In hierarchic trust models, the root CA issues subordinate CAs which can either also issue other CAs, or sign the certificates of end users. It allows an arbitrary tree-structure of CAs in which each parent passes on its trust to the child CAs. (b) In a flat model of trust, the root CA directly issues the end user certificates. No intermediate CAs exist.

3.2.4.2 Certificate Authorities (CA)

This *trust*-based mechanism also works on a large scale: *Certificate Authorities (CA)*, or *Certification Authorities*, are widely trusted entities that issue certificates of the entities in a PKI and make it possible to verify their identity. Since they are responsible for *certifying* the key pair/identity-binding of participants by their digital signature [28], they are an essential element especially in large PKIs.

In a typical public key infrastructure, only very few CAs are appointed so that most (or even all) of them are known and trusted by all identities. Participants of the network inherit their trust and are able to verify the authenticity of other users' certificates.

Just like normal certificates, CA certificates can be signed by other entities in order to transfer the trust from the CA to a user certificate. If CA_1, for instance, signs the certificate $Cert_A$ of a user A, other users are able to verify A's signature using the CA's public key certificate $Cert_{CA_1}$.

In a PKI, CAs are commonly organized in a *trust model*. That is, they are dependent of each other and structured, for instance, hierarchically or in a distributed architecture. A wide-spread approach is to organize CAs in a strict hierarchy with a *root CA* at the top of a tree and subordinate CAs below (fig. 3.3a). The leaves of the tree are generally called *end users* as they represent non-CA entities and cannot issue other certificates. Since no other certificate is classified higher than root CAs, they are commonly self-signed. Thus, the owner of the certificate (*subject*) and the entity that signed it (*issuer*) are the same.

A well-known environment with a hierarchical trust model appears in secure Internet connections. The HTTPS protocol combines plaintext HTTP communication with an underlying encryption layer. When first connecting to a secure Web server (indicated by

the schema *https://*), the user's browser starts a handshake procedure in which the server transfers its public key certificate. The browser then looks up the issuer-information of the certificate and tests whether it knows the corresponding CA or one of its superior CAs. If it does, it verifies the server's authenticity by probing its certificate against the CA. If this action fails or the CA of server certificate's issuer is not found, the browser displays a warning message and lets the user decide how to proceed. If the verification is successful, the handshake continues and the secure connection is commonly indicated to the user by a little closed lock in the status bar.

3.2.5 Standard Specifications

In order to be able to use PKI techniques and concepts in practice, various standards have been established over the years. This section briefly describes the ones used in the thesis' application.

3.2.5.1 X.509 Certificates

The Telecommunication Sector of the International Telecommunication Union (ITU-T), a "*United Nations agency for information and communication technology issues*" [29], developed a large suite of standards that includes various PKI related topics. The *X.500* suite of standards includes a specification for one of the most important parts of today's public key infrastructures: The *X.509 certificate* standard specification.

X.509 [30, 31] is a widespread certificate format used in most well-known PKI implementations. It consists of the subject's public key and various other fields such as the issuer's name, the subject's name and an expiry date. Furthermore, it contains a digital signature of the issuer over the whole certificate object. In their current third version, X.509 certificates are very open to extensions and allow great interoperability between various systems. All browsers and e-mail clients that support secure connections, e.g., come preinstalled with many root X.509 certificates of big CA companies such as VeriSign or Thawte. Figure 3.4 shows a self-signed root CA certificate of Thawte, Inc.

In order to provide a unique network identity for each user or system, the subject and issuer name are commonly structured according to the X.500 *distinguished names (DN)* standard [32, 33]. A DN is a string that represents an entity in a hierarchically structured environment. This could, for instance, be a person in a company, or a Web server in the university's data center.

A DN typically consists of *country (C)*, *state (ST)*, *organization (O)*, *organizational unit (OU)* and *common name (CN)*, but also supports user-defined key/value-pairs.

```
Certificate
    Version: v3
    Serial number: 0
    Issuer:
        CN=Thawte Basic CA, O=Thawte Consulting, L=Cape Town, ST=Western Cape, C=ZA
    Subject:
        CN=Thawte Basic CA, O=Thawte Consulting, L=Cape Town, ST=Western Cape, C=ZA
    Subject Public Key Info:
        Algorithm: PKCS #1 RSA Encryption
        Public Key:
           bc bc 93 53 6d c0 50 4f 82 15 e6 48 94 35 a6 5a
           be 6f 42 fa 0f 47 ee ...
    Validity: Mon Jan 01 01:00:00 CET 1996 until Fri Jan 01 00:59:59 CET 2021
    Extensions:
        BasicConstraints: [CA: true, Max. intermediate CAs: unlimited]
Certificate Signature:
    Algorithm: PKCS #1 MD5 With RSA Encryption
    Signature:
        2d e2 99 6b b0 3d 7a 89 d7 59 a2 94 01 1f 2b dd
        12 4b 53 c2 ad 7f aa a7 ...
```

Figure 3.4: A self-signed root CA certificate of Thawte, Inc. Exported from the Firefox preinstalled CA certificates.

In most secure protocols such as HTTPS or IMAPS, the *common name* field is used for hostname verification. That is, the server certificate's CN must match the hostname of the connection. If *ebay.com*'s Web server, for instance, would use a certificate issued to "CN=amazon.com, ...", the browser would alert the user because the hostnames do not match.

3.2.5.2 Transport Layer Security (TLS)

Transport Layer Security (TLS), successor of the *Secure Sockets Layer (SSL)*[4], is a cryptographic protocol that provides a secure channel between two communicating entities over which they can send arbitrary application data. It provides server *authentication*, *encryption* and *message integrity* and can also be used to authenticate the client. As of today, it is by far the most commonly used technology to realize a PKI.

TLS is designed for application independent encryption and is placed between the TCP layer and the application layer in the protocol stack. Due to the fact that it relies on TCP, but is somewhat invisible to the application protocol, it is used to provide security to old plaintext protocols such as FTP or NNTP. That means, "*any protocol that can be carried over TCP can be secured using SSL or TLS*". [33]

The protocol divides the connection in two phases. Before any application data is transferred, TLS needs to establish a secure connection with the remote host. This so called

[4]TLS is considered version 3.1 of the Secure Sockets Layer. Like most literature, this thesis uses both terms as synonyms.

handshake phase ensures that the server authenticates to the client and exchanges a key for future communication. After a successful handshake, client and server communicate using the negotiated symmetric key in the *data transfer* phase.

The handshake is the most important part of the protocol and fulfills three purposes [33]:

- Authentication: TLS enforces a correct certificate based authentication of the server to the client, but also enables client authentication.
- Cipher suite: Since TLS allows many different cryptographic algorithms and hash functions, client and server need to negotiate a *cipher suite* used for future messages.
- Key: Furthermore, they need to transfer or exchange the symmetric key used to encrypt the packets in the data transfer phase.

In order to meet the three requirements, the handshake is divided in several steps (fig. 3.5). The following listing describes the steps briefly:

Step 1-2 The client sends a list of available cipher suites and the server chooses the best that both of them support.

Step 3-4 The server sends its certificate chain, i.e. its own certificates, the CA that signed it as well as all superior CAs in the certificate hierarchy. If it wants the client to authenticate, it also sends a certificate request.

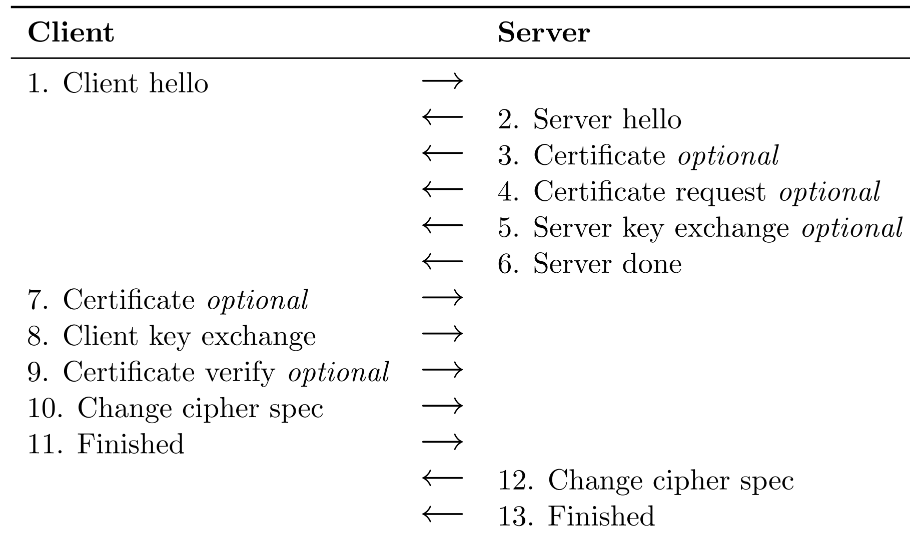
Step 5-6 The server sends a temporary key for the encryption of the later client key exchange; but only if its certificate does not provide enough information to use it for encryption, e.g., if it has a DSA public key. After this is done, it tells the client that it is finished with the initial negotiations.

Step 7 If the server requested a certificate in step 4, the client sends its certificate chain (just like the server did in step 3).

Step 8 The client generates the symmetric key for later communication and either encrypts it with the servers public key (RSA), or with the temporary key from step 5.

Step 9 If the server requested a client certificate, the client sends digitally signed information so that the server is able to verify its identity.

Step 10-13 Client and server tell eachother that they will use the negotiated key from now on and that the handshake is done.



Client		**Server**
1. Client hello	⟶	
	⟵	2. Server hello
	⟵	3. Certificate *optional*
	⟵	4. Certificate request *optional*
	⟵	5. Server key exchange *optional*
	⟵	6. Server done
7. Certificate *optional*	⟶	
8. Client key exchange	⟶	
9. Certificate verify *optional*	⟶	
10. Change cipher spec	⟶	
11. Finished	⟶	
	⟵	12. Change cipher spec
	⟵	13. Finished

FIGURE 3.5: TLS handshake phases [34].

The most important step of the handshake is step 8 as it generates the symmetric key for future encrypted messages. The rest of the handshake simply verifies the identity of the participants and prepares the key exchange.

Chapter 4

Application Design

The previous chapters mainly focused on getting to know the approaches of other related systems and providing a basis for the following discussion about possible application designs. This chapter develops the basic idea of the system approach by defining the system requirements in order to derive the most suitable design.

The first section focuses on formulating the application's purpose and its specifications, and will discuss design related topics. The second part points out why the system has been designed as present and demonstrates the results of the process. The third part finally introduces the developed system.

4.1 Initial Situation and Goals

Before debating the pros and cons of the different possible approaches, the goals and requirements must be defined in order to evaluate various ideas and the later results.

The obvious objective of the thesis' application is to create a URL blacklisting service similar to the ones introduced in chapter 2. However, it aims to fulfill several other functions and intends to create an extendable system which can be used for various, possibly completely different purposes.

The following paragraphs give an impression of the intended purpose.

4.1.1 Blacklist Service

The Laboratory of Dependable Distributed Systems at the University of Mannheim is currently running several automated systems to identify malicious Web sites on the

Internet. For this purpose, they set up a highly controlled network of *honeyclients*, called *honeynet*, which crawls the Web and searches for Web pages that try to attack users or infect them with malware via drive-by downloads. The network collects information about different kinds of Web-related attacks using low- and high-interaction honeyclients on the one hand, and Web-decoys on the other hand [35].

In order to use the gathered data not only for study purposes, but make the results available to a broad audience, this thesis' *first goal* is it to provide a service with an application programmable interface (API) for querying the results of the honeynet's analysis.

Using the Honeynet-provided Data

To reach this goal, the following general requirements need to be considered:

- Large Amounts of Data: The honeynet provides an extensive amount of data. That is, the application must be able to handle many records.
- Concurrency: The service shall be available for a large number of users. Therefore, it needs to be able to handle many requests at the same time and provide a fault-tolerant, redundant infrastructure.
- Open API: Even though the intended client form is a browser plug-in, the service shall be usable by any application through the provided API.
- Secure Communication: The user-queries might include confidential data (in the URL). Hence, the communication between client and the service needs to be encrypted using standard protocols such as TLS/SSL. This ensures that a broad range of clients is able to communicate with the server.
- Fast Queries: Clients might query the service very often to request information about URLs or domains. A fast and lightweight request/response *round-trip time (RTT)* is hence desirable.
- Admin Interface: Due to the fact that the honeyclients constantly collect new data, the API needs to provide functions to add, alter and remove items from the database of the service. Since only they shall be allowed to alter the data, an authentication-method is required to prove their identity.

Scalability, Extendibility and Reliability

The suggested service holds great potential since it provides obvious benefits to end-users. This intends that the possibility of a growing user-base must be considered in

the design process. Thus, it is important to design the service from the beginning in a way that it can be scaled very easily, is extendable and provides a certain amount of reliability.

- Scalability: If the service reaches its peak in terms of resource usage, it must be possible to easily add resources such as disk space, memory or CPU time without shutting down the system.
- Extendibility: A service typically evolves over time which makes it desirable to be able to change the protocol and extend it with new versions. The underlying system hence should be designed in a general matter to enable version changes.
- Reliability: Since the base system has to handle big amounts of requests and clients may rely on its service, it must be able to cope with server failures and provide high availability.

Encryption and Attack Resistance

Every globally available service faces the usual threats of Internet applications, including, for example, Distributed Denial of Service (DDoS) or Man-In-The-Middle attacks. Therefore, the system aims to provide secure communication on the one hand, and a certain amount of attack resistance on the other hand.

- Encrypted Communication: As indicated above, the communication between client and the servers of the system needs to be encrypted using standard protocols. Moreover, if the service implements a distributed infrastructure, the entities of the system also have to communicate with each other. Therefore, communication encryption between the servers is also required.
- Attack Resistance: Just like Internet technologies evolve over time, the number of possible attacks grows constantly. Due to these facts, it is crucial for an application like this to provide at least some amount of attack resistance. Basic approaches to realize this include the provision of redundancy and availability.

4.1.2 P2P-based Core System

During the evaluation of the system's properties, it became clear that at some point in the design process one had to choose how to realize the core system of the service.

While some of the above requirements are relatively easy to achieve with any underlying system, most of them require a special infrastructure. Reliability, fault-tolerance and

attack resistance, for instance, cannot be achieved using only a single server, but require multiple computers to work together. There are several ways to fulfill these requirements – be it a highly synchronized server environment, a P2P-based network, or a virtualized infrastructure.

After a brief evaluation of the possibilities and a comparison between the approaches, a P2P-based network has been chosen to serve as the core system. On the one hand, this decision is based on the fact that the properties of peer-to-peer systems mostly match the ones needed for the blacklist application. On the other hand, this approach has experimental character and is supposed to determine the suitability of the application to the P2P-approach.

Thus, the *second goal* of this thesis is to implement a P2P-protocol and extend it to match the requirements of the application. In particular, this includes the design of a *secure and trusted DHT protocol* which can be used to store any kind of data in a distributed environment, and implements the basic concepts of a public key infrastructure. That is, the new DHT combines the scalability and flexibility of a P2P-based infrastructure with the security of a PKI.

The basic idea is to create a trustworthy P2P network, in which all communication is encrypted and every peer has to authenticate before it can join. Especially the latter property represents a rather unusual approach, since almost all P2P/DHT protocols are open for everyone and allow peers to join and leave the network as they like. The proposed secure DHT protocol has a different objective: It aims to use the openness of P2P systems for an easy resource management, but restricts the access to the network to authenticated peers only. Combining these two properties creates a very extendable distributed, secure and access-restricted core system.

The requirements of the core system mostly match the ones of the blacklist service. Especially the DHT- and security-related topics, e.g. scalability and secure communication, need to be considered while designing the protocol. However, it is essential to point out several other prerequisites in order to allow a later evaluation of the resulting implementation.

Restricted Network Access

In contrast to usual P2P networks, the new DHT shall restrict the access to a known set of peers. Therefore, each node needs a unique verifiable identity in order to join the network.

- Node Identity: Every peer that likes to participate in the network needs to identify itself before joining. That is, each node needs a unique, provable identity to be part of the DHT.

- Restricted Access: In order to prevent the network from flooding attacks (such as the Sybil attack in Kademlia), nodes attempting to join the network need to prove that they are allowed to do so. Hence, the network needs to provide some sort of authentication mechanism to verify joining nodes. If a node cannot authenticate, it is not allowed to be a part of the network.

Basic PKI Support

The new DHT protocol aims to realize at least the basic concepts of a public key infrastructure. That is, it must fulfill the requirements of a PKI like described in section 3.2.4 (also cmp. [26]):

- Authentication and Identification: The identity of each node in the network needs to be assured and verifiable by other entities (node identification and authentication). Furthermore, the origin of a message needs to be verifiable by the recipient-node (data origin identification).

- Integrity: Each node of the DHT needs to be able to prove that a message has not been altered by a third entity, or must be able to detect alteration, respectively.

- Confidentiality: No entity of the network is able to read messages between two nodes, except the sender and recipient of the message.

4.2 Design Discussion

After specifying the requirements of the URL blacklisting service and the secure P2P network, the actual design can be elaborated. In order to achieve the best result, the following sections discuss possible alternative approaches and introduce the basic architecture.

As the related approaches from chapter 2 demonstrate, a service like this could be realized in many different ways. Hence, designing an efficient and extendable framework is not an easy task.

4.2.1 Peer-to-Peer as Underlying Technology

Probably the most relevant decision is choosing a suitable underlying core technology. As already indicated in the previous section, P2P has been chosen to serve as the foundation on which the entire system is built upon.

At first view however, the client-server model seems to be the more plausible solution: It has been around for quite a long time and nearly every programming language ships with several ready-made packages. Implementing a server is a job of only a few minutes and even PKI concepts are mostly supported natively (TLS/SSL). Thus, developing a service on client-server basis accelerates the programming process and minimizes the required design time of the project for small scale applications. However, as soon as the service starts to grow and resource usage reaches its peek, additional technologies are necessary to keep it running, or even expand it to a larger scale.

Commonly used approaches to scale client-server based systems are replication and synchronization: To allow a greater number of users, a service does not provide only one server process, but starts multiple identical servers on different physical machines. Even though this solves resource- and performance-bottlenecks, it requires complicated data replication and synchronization technologies.

In contrast to the client-server model, the concepts of P2P aim to provide a scalable environment without having to cope with synchronization issues. That is, P2P is scalable *by design* and does not need complicated additional software to work on a large scale. Distributed hash tables in particular already include a number of desirable attributes such as scalability or resource flexibility.[1] And even though the implementation of P2P-based systems is considerably more difficult than developing a client-server framework, it sometimes is the more suitable solution for specific problems.

In order to determine the suitability of P2P to this type of service, it has been chosen as the underlying architecture.

4.2.2 Kademlia as DHT Protocol

Chapter 3.1.3 already introduced some of the most important distributed hash tables and mentioned that Kademlia has been chosen to serve as basis for this thesis' application. However, due to the similarity of all DHT protocols, this section briefly discusses why Kademlia is the best solution for the proposed service.

[1]cmp. section 3.1 for a detailed description of P2P attributes

- Performance: In terms of its routing algorithm, Kademlia is very similar to Pastry: Both focus on consistency and performance, and forward messages to nodes that share the longest prefix with the target identifier [22]. However, Pastry's routing metric is adjustable and does not always correspond to the closeness of identifiers. "*As a result, Pastry requires two routing phases which impacts the performance*" significantly [13]. In order to avoid discontinuities between different metrics, Kademlia optimizes Pastry by using a single routing algorithm from start to finish, and outperforms it in terms of lookup time and consistency.

- Routing Strategy: Another major difference to other DHTs is that Kademlia uses the fact, that "*nodes with a long uptime have a higher probability of remaining available than fresh nodes*" [13]. That is, Kademlia's routing strategy increases the stability of the network by favoring older nodes over newly joined nodes. This prevents the network from being flooded by attacker nodes on the one hand, and provides a certain resistance against DDoS attacks on the other hand.

- Concurrency: Kademlia is one of only few DHTs that support parallel lookups by design. Using the special α parameter, the network can be configured to ask α nodes concurrently for its FIND_NODE- and FIND_VALUE operations. This effectively implements asynchronous routing and lets users trade their bandwidth for "*lowest-latency hop selection and delay-free fault recovery*" [24]. Thus, the greater α is, the faster is the lookup algorithm, but the more bandwidth will be used.

In conclusion, Kademlia provides several features that none of the other DHTs offer. It has a fast routing algorithm and provides a stable network by exploiting the fact "*that node failures are inversely related to uptime.* Furthermore, it minimizes the number of configuration messages with its single phase routing algorithm, and uses "*parallel, asynchronous queries to avoid timeout delays from failed nodes*" [24].

For these reasons, Kademlia is an ideal solution to serve as the underlying P2P protocol for the proposed system.

4.2.3 PKI for Peer-to-Peer

The idea of P2P arose from the need for decentralized and self-organized computer networks that can be scaled by simply adding new nodes to the system. *Confidentiality* and *authentication*, however, were never a requirement for P2P systems, and even today's well-known networks (such as eDonkey or BitTorrent) are mostly based on plaintext

communication between unauthenticated nodes. And even though some of them allow data encryption of some sort, the majority of the P2P traffic is still unencrypted.

Supposedly the primary reason for this lack of secure communication is that most networks have been designed for a completely different purpose: The majority of P2P protocols is open for everyone and do not require any form of authentication. That is, they allow everybody to join the network without having to authenticate first, so that any client can store or retrieve information. Because of this openness of all content, communication encryption is pointless for most scenarios. If an attacker, for instance, attempts to retrieve the key K, he or she does not have to eavesdrop the communication between two communicating peers because K is available to anyone. Therefore, A can simply join the network and retrieve K like a normal client.

In closed networks, however, authentication and communication encryption become absolute requirements in order to provide a trustworthy P2P network. Public key infrastructures provide these properties and are hence a qualified solution to realize a closed, secure P2P network:

- Authentication: In contrast to open networks, closed systems only allow a given set of peers to participate in the network. In order to control which peers are allowed to join, the system has to provide some sort of authentication. PKI is ideal for this purpose because it provides certificate-based authentication by design: Equipped with a CA-signed public key certificate, a *trusted peer* is able to prove its identity to the network before joining. Similar to the TLS handshake (cmp. figure 3.5), this initial procedure between a joining node and the bootstrap node simply verifies the CA's signature (from the joining node's certificate), and proves that it can read public key encrypted data with its private key.

- Confidentiality: Secure communication in open networks is pointless because the content is available to every client. In closed networks, however, the encryption of communication is required to prevent eavesdropping from unauthenticated peers or Man-in-the-Middle attacks, respectively. PKIs enable data encryption using public key cryptography and are hence suitable to provide confidentiality.

Even though public key infrastructures seem to fit perfectly for a secure distributed hash table, the solution raises many questions, especially in terms of road capability and performance. In order to determine the practicability of PKI for the use in P2P networks, this thesis uses PKI concepts for its implementation.

4.2.4 Secure Extension for Kademlia

Kademlia, the chosen DHT protocol, does not facilitate any PKI concepts by design. Therefore, it has to be extended in order to fulfill the requirements of the core system and the secure DHT.

At first view, SSL/TLS (cmp. section 3.2.5.2) is suitable to meet the demands since it allows client authentication and enables transfer encryption between communicating nodes. In practice, however, the structure of SSL/TLS is too heavy and its handshake too slow to provide fast communication in a P2P network:

Slim Handshake

Kademlia and other DHT protocols communicate with many different nodes and even for a single FIND_NODE-query, it might be necessary to connect to several so far unknown nodes. In open P2P networks, connecting several peers does not affect the overall performance significantly because the communication is typically based on the lightweight User Datagram Protocol (UDP). In a secure network, however, nodes need to verify each others identity with a handshake before they can communicate securely. Since this procedure is mostly TCP-based and takes much longer, a fast handshake is essential.

The current version of the SSL/TLS handshake (cmp. figure 3.5) includes operations that are not necessary for the proposed core system and slow down the lookup significantly. The cipher negotiations (step 1-2), or the extensive finalization communication (step 10-13), for instance, are useless if the P2P protocol defines specific encryption algorithms.

Thus, the secure version of the Kademlia protocol, which has been developed within the scope of this thesis, does not use the SSL/TLS framework for the communication between nodes, but uses its very own slim handshake: Instead of supporting various different cipher suites and algorithms, the slim handshake predefines the algorithms in order to minimize the overhead and speed up the process. After a successful initial authentication and the key-exchange, the protocol furthermore enforces fast connectionless communication between network participants using symmetrically encrypted messages.

Flat Trust Model

Certificate authorities in PKIs are typically structured in trust models (cmp. chapter 3.2.4.2). CAs in SSL/TLS-based systems, for instance, are organized hierarchically and form a tree-like structure of CA-certificates. This allows very flexible inheritance of trust throughout the network on the one hand, but slows down the handshake on the other hand: That means, if the certificate of an entity has been signed by a sub-sub CA of the

root CA, e.g., the SSL/TLS handshake dictates, that the whole certificate chain has to be transferred to, and verified by the opposing party. That is, instead of simply sending one certificate, the entity has to transfer its own certificate as well as all superior CA certificates. The other entity then has to verify each of the certificates in the certificate chain to prove the identity of the user.

In order to avoid this scenario, the thesis' protocol does not use a hierarchical trust model, but relies on one single root CA without any possible child authorities. Thus, each node in the network has to possess a certificate directly signed by the root CA. This flat trust model accelerates the handshake, because nodes need to transfer and verify only one certificate instead of the whole certificate chain.

4.3 Basic Architecture

After discussing possible design directions and having figured out the best approach, the following section introduces the basic architecture of the system.

4.3.1 Overview

As briefly described in the above sections, the blacklisting service is based on a P2P network of interconnected nodes. Each node is an equal part of a distributed hash table and only stores the blacklist entries it is responsible for. The application splits in two major parts: The core network and the blacklisting service.

While the Kademlia-based core network, called *KadS*, implements a trusted DHT and acts as data store, the blacklisting service is built on top of the DHT and simply uses the distributed storage.

As figure 4.1 shows, the core network consists of several *KadS nodes*. Each node stores a small part of the DHT in its local hash table and keeps track of other nodes in the local routing tree. In order to make the DHT accessible, it furthermore provides an API to add, alter and delete entries.

The KadS network only provides a P2P-based storage environment, but does not validate the data it stores. For the blacklist service, however, a fixed data-structure is required: The *blacklist nodes* represent the actual user-accessible service and solve this problem by enforcing a business-logic as well as a fixed data structure for the hash table values. Each node encapsulates a KadS node and implements a secure TCP/UDP server to communicate with browser- and honeyclients.

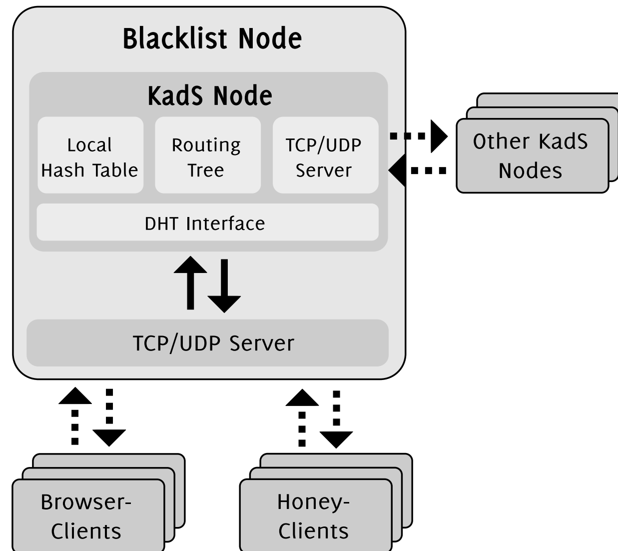


FIGURE 4.1: Schematic design of a blacklist node: The KadS nodes form the basis for the network and store the blacklist information. Blacklist nodes extend the core system and provide publicly available interfaces for the clients.

The following sections describe both the core network and the blacklisting service in greater detail.

4.3.2 KadS: Core Network

The core network is an access-restricted, secure distributed hash table. Its design is based on a modified version of the Kademlia DHT protocol and extends it to a fully encrypted public key infrastructure. While the basic operations are almost identical to Kademlia, KadS restricts the access to the network on the one hand, and enforces nodes to encrypt their communication on the other hand. In order to do so, nodes need to authenticate in a handshake procedure before DHT-related messages can be exchanged. Only if they successfully verified each other's identity, they are able to exchange messages using a symmetric session-key.

Thus, the major differences of KadS to the Kademlia protocol are *access restriction* and *communication encryption.*

Access Restriction: The TCP-based KadS Handshake

While Kademlia allows every peer to join the network, KadS restricts the access and enforces a handshake when two nodes first meet. That is, each time a node likes to communicate with another node in the network, it has to prove its identity before performing any DHT operations. To realize this, KadS uses certificate-based authentication

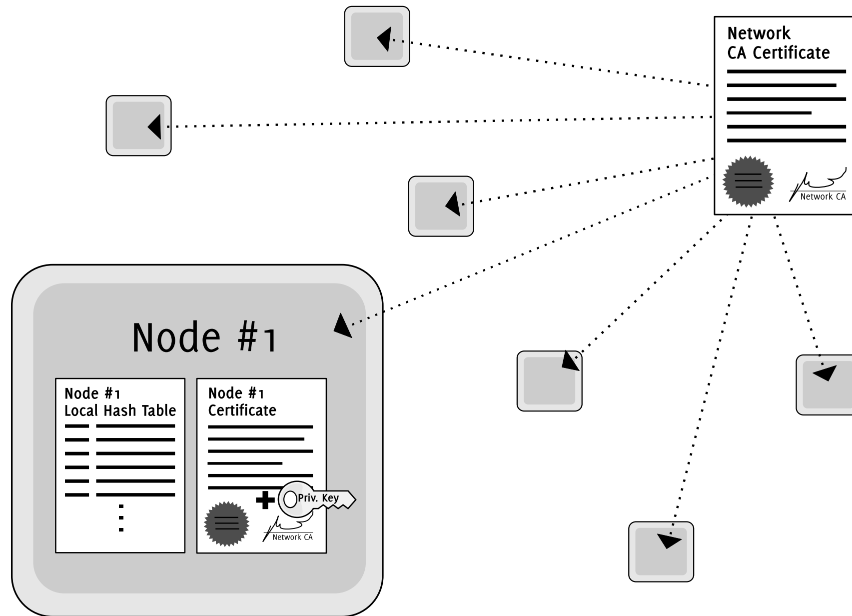


FIGURE 4.2: The network provides a single root CA which signs the certificates of all participating nodes. Each node contains (1) a local hash table to store the DHT entries it is responsible for, (2) a public key certificate signed by the network CA and a corresponding private key in order to encrypt/decrypt messages, (3) and a copy of the network CA certificate itself to verify the identity of other nodes.

and equips each node with several PKI-related elements. In particular, each node owns a copy of the network's CA certificate and a distinct CA-signed key-pair:

- CA Certificate: The KadS network implements a flat trust model (cmp. figure 4.2) and therefore provides a single CA certificate which signs each of the nodes' certificates. Every node that likes to participate in the network needs to be in possession of a valid certificate, signed by the network's certificate authority. If it cannot provide a CA-signed certificate, it cannot join the network.
- Node Key-pair: Each node is equipped with a unique private key and a corresponding CA-signed public key certificate. The certificate represents the node's *provable identity* needed for authentication. A short representation of the identity, the node ID, derives from the node certificate using the SHA1-function over the public key bit-stream.

The KadS handshake verifies the identity of nodes attempting to communicate, and exchanges a session-key for future DHT-related messages. Similar to the SSL handshake, the KadS handshake is TCP-based and follows likewise steps: First, the two peers exchange their public key certificates to make sure that both are allowed to be part of the network. By comparing if the network CA really signed the other peer's certificate, both nodes are able to prove each other's identity. If the identity-check succeeds, the

two nodes exchange randomly generated data and create a secret-key. This symmetric key will be used for future messages between the two nodes.

Schematically, the handshake follows these steps[2]:

- The nodes exchange public key certificates
- Both verify the other node's identity with the CA certificate
- Both exchange randomly generated data and create a secret key
- Further messages are encrypted using the symmetric key

Kademlia does not encrypt any connection at all and hence has no need for a handshake procedure. KadS however requires this handshake in two situations:

- Joining Node: When a new node joins the network, it needs to authenticate with the bootstrap node first before it can communicate with it.
- FIND_*-operations: Kademlia's FIND_NODE-operation returns the internet socket addresses of the closest nodes to a given identifier. Following the protocol, some of the returned nodes have to be queried as well to get the final result. But before the local node can send them any DHT-related messages, it has to verify their identity using the handshake operation. The FIND_VALUE-operation behaves analogue.

Communication Encryption: UDP-based DHT Messages

After a successful handshake, two nodes have exchanged a symmetric secret key and can perform the usual Kademlia operations, i.e. FIND_*, STORE and PING. In fact, once the two nodes know each other, their communication is identical to the one in a Kademlia network: Like Kademlia, KadS uses UDP to send and receive RPCs, is based on the XOR metric, and relies on concurrency to accelerate its lookup algorithms.

However, in contrast to the original protocol, KadS encrypts every RPC using a secret key, so that confidentiality is guaranteed. In fact, every pair of nodes in a KadS network uses a different secret key, so that only the sending and the receiving nodes are able to read the message.

Example

Consider the following example: A node A likes to join the network in order to lookup the value of key K. It knows several other nodes that are already part of the DHT, e.g.

[2] A detailed description of the KadS handshake will be given in chapter 5.1.4

B, C, etc. After choosing a bootstrap node, for example B, A needs to perform the following operations to retrieve the desired value for K:

1. Perform a handshake with B: Establish a connection, verify B's identity and exchange session-key SK_{AB}
2. Perform a FIND_NODE(A)-operation in order to populate the routing tree
3. Perform a FIND_VALUE(K)-operation to locate the desired key and retrieve its value

The first two steps are part of the entry protocol of Kademlia and KadS, respectively: In a Kademlia network, a joining node simply connects to the bootstrap node and performs a lookup on itself. In this example, A would connect to B and query it to return the closest nodes around A, i.e. A effectively would get to know its neighborhood in the network.

In a KadS network, A first has to shake hands with B to win its trust (step 1). After successfully authenticating each other, A gathers information about the closest nodes around itself. But in contrast to Kademlia, A not only receives the addresses of them, but has to shake hands with some of them to win their trust as well (step 2). After entering the network and having proven the identity to the neighbors, A can finally perform a lookup on the desired key K. Similar to the Kademlia operation, A iteratively queries the closest nodes and learns new nodes with every iteration. But unlike Kademlia, KadS nodes need to authenticate with newly received nodes in order to query them as well (step 3).

4.3.3 Blacklist Service

As figure 4.1 shows, the blacklist service is wrapped around the core system and uses its interface to store and retrieve blacklist entries. In fact, it simply uses the secure DHT protocol as database and enforces a specific data-type for the hash table keys and values. To make the blacklist entries available to clients, it furthermore provides two different interfaces for browser plug-ins and the honeyclients. That is, the underlying core system provides almost all functionality while the actual blacklist service only uses its infrastructure for secure distributed data provisioning. In fact, the core system could be used for several different purposes at the same time as long as the on-top services such as the blacklist service generate differentiating hash table keys.

Communication with the KadS network

In order to receive and/or store entries in the DHT, each blacklist node needs to be connected to the KadS network. For this purpose, each of them encapsulates a KadS node and additionally provides outside interfaces for the blacklist clients. That is, the actual blacklist service does not provide any DHT-related functions, but simply uses the KadS node's methods to access the distributed hash table. Querying the network for the blacklist entry of the domain *example.com*, for instance, is nothing more but a simple call to the KadS node's `get`-method.

TCP/UDP-based Interfaces

To make the blacklist service accessible to the outside clients such as the honeyclients and the browser plug-in, each blacklist node provides two interfaces: While the UDP-based query service provides a fast way to retrieve information about domains, the TCP-based service is responsible for anonymously logging in browser-clients, and for authenticating honeyclients, respectively. That means, before a client can use the fast UDP service, it has to exchange secret keys using the TCP service.

The primary interface for all clients is the TCP-based blacklist server. It represents the first contact point for any client that likes to interact with the service. It implements a simple SSL/TLS server and interacts with clients via a self-described protocol.

In contrast to the later introduced UDP-based query server, the TCP server supports the whole set of functionality. Its main purposes are:

- Node Authentication: All clients connecting to the blacklist node need to verify their identity before communicating with it. Especially since they might transfer sensitive information to the node, server/node authentication is crucial.
- Client Authentication (only Honeyclients): In order to be able to store information in the network, clients are able to authenticate themselves using a CA-signed public key certificate. If a client successfully authenticates in the SSL session, it is able to store blacklist entries.
- Key Exchange: Since the UDP-based query server only allows encrypted communication, each client needs to retrieve a session ID and a secret key from the blacklist node in order to encrypt future UDP messages.
- Retrieve/Store Blacklist Entries: After a successful SSL handshake and key exchange, clients are able to send commands to the SSL socket. Depending on what access they have been granted, they can store entries in the network, or just query it for domain information.

The second available service is the UDP-based query server. Unlike the blacklist server, it focuses on the provisioning of blacklist data only and requires a previous login at the TCP server. Instead of having to go through the 3-way handshake and several acknowledgement messages of TCP, the query server uses UDP for lightweight messages and fast query roundtrip time. Due to the fact that the query server skips heavyweight protocol overhead, it is intended to be used by the browser plug-in rather than the honeyclients.

Entry Structure

The blacklist service enforces a fixed data structure for the DHT entries. While the key of each entry is a simple SHA1 hash over the domain string, the value is a complex data structure and contains the following content:

- Domain: The domain name for the analysis entry.
- Expiry Date: Since most malicious Web sites only exist for a few days, each entry expires at some point.
- Analysis Code: An integer code to quickly identify the analysis status of the domain. In particular:

  UNKNOWN → Nothing known about the domain.

  FAILED → The lookup either timed out or failed for some other reasons.

  CLEAN → The whole domain seems to be clean, i.e. no reported infection.

  PARTIALLY_INFECTED → Some paths of this domain are infected. The attached list shows which paths are affected.

  INFECTED → The whole domain has been flagged as malicious.

- Path List: An optional list of domain paths or URLs that specify malicious parts of the domain.

Both TCP- and UDP-server return this structure to the clients when queried for the malignity of a domain. To store entries in the network, a honeyclient has to pass a structure like this to the blacklist server.

Chapter 5

Implementation

This chapter introduces the concrete Java implementation of the KadS protocol and the blacklisting service. It will look into details on how the Kademlia protocol has been realized and demonstrate key functions of the system. The first part focuses on KadS, while the second part describes the URL blacklisting service.

5.1 Core Network: The Secure Peer-to-Peer Protocol

The core network consists of many interconnected KadS nodes. Each node represents a trusted identity of the network and implements the KadS protocol. The following sections introduce the essential parts of a KadS node and demonstrate how the peers communicate with each other.

5.1.1 Terminology

In order to establish easier understanding for the implementation, the list below describes the most important terms. Each of the described terms also relates to an existing Java class of the KadS core system.

- `Identifier`: Each node as well as each hash table entry in a KadS network is equipped with a 160-bit identifier. For KadS nodes, the *node ID* serves as short representation of the node's identity and derives from its public key certificate. For DHT entries, the identifier is commonly called *key* and is used to find a value in the hash table. Due to Kademlia's network design, both node ID and hash table key are directly related to the logical position in the network.

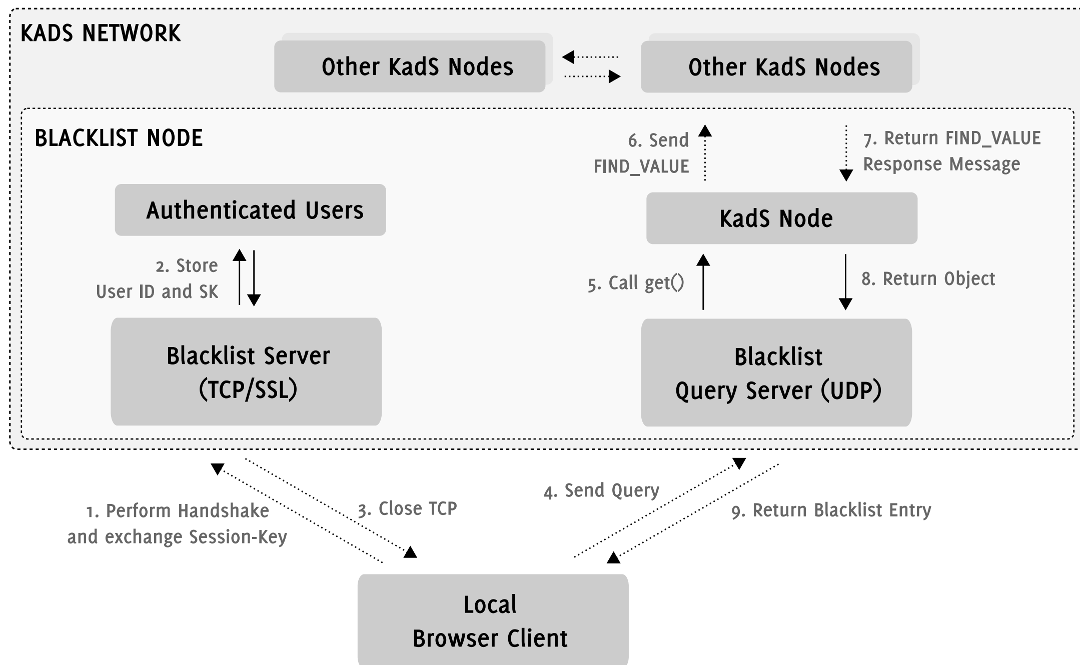


FIGURE 5.1: Before communication via the lightweight UDP service (step 4-9), the browser client needs to sign on at the TCP/SSL blacklist server (step 1-3). Once the browser exchanged a session key (SK), it can send encrypted UDP messages to the query server. The query server unpacks and decrypt each packet and queries the KadS network.

- `Identity`: KadS uses public key certificates to identify entities in the network. Each participating node needs to possess a CA-signed certificate to join. An *identity* is a X.509 certificate and its deriving identifier.

- `LocalIdentity`: In addition to the public key certificate, a *local identity* also carries the corresponding private key. That is, it extends the *identity* and consists of a CA-signed certificate, the deriving ID and the matching private key.

- `Node`: In this implementation, the term *node* does not relate to the provable identity of a node, but only represents the information required to locate it in the network. That is, a node consists of the node ID, its IP address and the TCP/UDP port number.

- `VerifiedNode`: Unlike the term node, a *verified node* does not only contain routing-related information, but also carries the node's identity and a negotiated session key. In order to send DHT-related messages, the local KadS node has to learn the identity of an unverified node and exchange a secret key. After shaking hands, a node therefore becomes a verified node.

5.1.2 Core Interface `DistributedHashMap`

The main purpose of the core network is to create a distributed hash table. In order to allow programs to access the data, a DHT has to provide several functions to add and remove items. The KadS network is based on the following simple API, similar to Java's `Map`-interface:

```
public interface DistributedHashMap {
  public void connect(InetSocketAddress address);
  public boolean contains(Identifier key);
  public Serializable get(Identifier key);
  public void put(Identifier key, Serializable value);
  public void remove(Identifier key);
  public void close();
}
```

Listing 5.1: Each KadS node implements the core interface DistributedHashMap.

While all data-related functions such as `contains` or `get` are self-evident and also appear in the Java `Map`-interface, `connect`- and `close` are DHT-specific methods. The `connect`-method initializes the connection to an existing network and `close` leaves the network and shuts down the local node.

5.1.3 Local `KadS` Node

KadS provides an implementation for the `DistributedHashMap`-interface: The `KadS` class represents a single node (cmp. figure 4.2) and mainly consists of the following components:

- `Config`: The constructor of the `KadS` class requires certain configuration settings to be present. The `Config` class represents the config-file of a single KadS node. It allows adjusting various settings (such as the ports, timeouts, etc.) on the one hand, but also possesses the CA's `Identity` and the node's `LocalIdentity`.
- `RoutingTree`: Each KadS node keeps track of some of the network's nodes using a binary `RoutingTree`. This tree is designed to know many nodes in the direct neighborhood, but fewer farther nodes. Like in Kademlia, KadS' routing tree is almost identical to the original design. In contrast to Kademlia, however, the KadS routing tree only keeps `VerifiedNode` objects.

- `HandshakeClient`: Before the local KadS node can send any Kademlia RPCs to a node, it has to verify its identity. The `HandshakeClient` shakes hands with an unverified `Node` and adds a corresponding `VerifiedNode` to the `RoutingTree`.
- `HandshakeServer`: The TCP-based `HandshakeServer` is the complement of the `HandshakeClient`. It accepts handshake requests, by default on TCP port 6852, and performs the same operations as the client.
- `Messenger`: After successfully verifying the identity of a node, KadS nodes can exchange DHT-related messages. The `Messenger` combines a UDP server (also port 6582) and client and is responsible for handling request and response messages.

5.1.4 Initial Authentication: `HandshakeClient` and `HandshakeServer`

Like briefly described earlier, the KadS handshake is very lightweight. While SSL allows several cipher suites and supports many different algorithms, KadS predefines the cipher suite and strips other unnecessary actions in order to accelerate the negotiations. It defines the following algorithms:

- Certificates/Authentication: Public and private keys of the KadS network used in this thesis are 1024 bit RSA key-pairs. Both CA certificates and node certificates exclusively use this algorithm for authentication. In particular, the public keys are used during the handshake procedure to exchange the secret key. The signature algorithm of the certificates is SHA-256 with RSA.
- Symmetric Session Key: For encrypting the UDP packets in DHT-related messages, KadS predefines the use of the Advanced Encryption Standard (AES), 128 bit. Before transferring a message, KadS serializes the message and encrypts it blockwise using AES and the negotiated session key. Since AES processes entire blocks at a time, it must be padded if there are not enough bytes to fill the last block. KadS uses the PKCS5Padding in ECB mode [36].

The two classes that mainly use the algorithms are `HandshakeServer` and `HandshakeClient`. Both are complementary classes responsible for handling the initial negotiations. They implement the KadS handshake and use the thread pool pattern to serve many nodes at the same time.

Handshake Protocol

The KadS handshake (fig. 5.2) is very similar to the SSL handshake, but reduces some unnecessary operations. It divides in five major phases:

	Node C (acting as Client)		Node S (acting as Server)
1.	$\texttt{send}(Hello, Version, Cert_C)$	$\longrightarrow$	$\texttt{verify}(Cert_C \text{ with } Cert_{CA})$
2.	$\texttt{verify}(Cert_S \text{ with } Cert_{CA})$ $SK_C := \texttt{random}(\text{64-bit})$	$\longleftarrow$	$\texttt{send}(Hello, Version, Cert_S)$
3.	$\texttt{send}(Pub_S(SK_C))$	$\longrightarrow$	$SK_C := Priv_S(Pub_S(SK_C))$ $SK_S := \texttt{random}(\text{64-bit})$ $SK := \texttt{concat}(SK_C, SK_S)$
4.	$SK_S := Priv_C(Pub_C(SK_S))$ $SK := \texttt{concat}(SK_C, SK_S)$ $\texttt{verify}(\text{received } SK_C)$	$\longleftarrow$	$\texttt{send}(Pub_C(SK_S), SK(SK_C))$
5.	$\texttt{send}(SK(SK_S), SK(port_C))$	$\longrightarrow$	$\texttt{verify}(\text{received } SK_S)$

FIGURE 5.2: KadS handshake: Unlike the heavyweight SSL handshake, the initial KadS negotiations are very light. Both nodes exchange certificates, verify each other's identity and pick a secret key for future UDP messages. After a successful handshake both nodes store the identity in their local routing tree together with the negotiated symmetric key.

In the first two phases (phase 1-2), client and server simply exchange their public key certificates and verify that the opposing certificate has been signed by the network's CA. Even though this seems to be a complete identity verification, it only proves that a node is in *possession* of a CA-signed public key certificate. However, it does not prove that the node really *is* this entity.

In order to substantiate the identities of both nodes, each of them needs to satisfactory show that the opposing node also possesses the corresponding private key. Only if a node can read a public key encrypted message, one can truly believe that it really is the claimed node.

Therefore, the handshake does not simply transfer the symmetric session key (SK) from one node to another, but assembles it from public key encrypted data: In the last three phases (phase 3-5) of the handshake, each node creates a random 64-bit byte-stream, encrypts it with the opposing public key and sends it to the other node. By concatenating the two decrypted random data streams, each node is able to create the 128-bit AES session key. To finally verify the identity of the other party, each node sends the opposing SK-encrypted part of the session key back to the other node. After comparing the own SK part with the received one, each node can verify the real identity of the opposing node.

Concurrency via Thread Pools

Each KadS node has to communicate with many other network nodes at the same time. To ensure a continuous service, it needs to process many messages and handshakes simultaneously. For this purpose, both `HandshakeServer` and `HandshakeClient` use the *thread pool* pattern: Thread pools are an effective mechanism to perform independent asynchronous tasks. A pool typically provides N task slots and can run multiple threads at the same time.

```
while (!serverSocket.isClosed()) {
  ...
  Socket clientSocket = serverSocket.accept();
  workerThreadPool.execute(
    new HandshakeServerWorker(clientSocket, ...) );
  ...
}
```

Listing 5.2: The handshake server passes connections to worker threads.

In this particular case, the `HandshakeServer` consists of a single TCP server socket and creates a new `HandshakeServerWorker`-thread for each incoming connection: The worker thread is then started by the thread pool and freed as soon as it terminates. The actual TCP server simply passes the connection to a worker and therefore only needs very little time for each client. Using this pattern, the handshake server can process N clients concurrently without having to refuse connections (listing 5.2).

Even though the `HandshakeClient` does not have to accept connections from other clients, it might be used by many local threads. To allow all running KadS operations to use it, the handshake client also uses a thread pool and creates `HandshakeClientWorker` objects to do the actual work.

```
public interface HandshakeFinishedListener {
  public void onHandshakeFinished(
    InetSocketAddress address, VerifiedNode newNode);
  public void onHandshakeFailed(InetSocketAddress address);
}
```

Listing 5.3: The handshake client notifies the registers listener as soon as the handshake is complete.

For the handshake server object, it does not matter when the handshake is finished because there are no subsequent operations. For any client operation, however, it is

crucial to know when the handshake terminates. Thus, the `HandshakeClient` additionally allows registering a `HandshakeFinishedListener`: As soon as the client worker finishes the handshake, or it fails for some reason, the according listener-method is called asynchronously (cmp. listing 5.3).

5.1.5 Messaging System: `Messenger` (UDP)

As soon as the local KadS node successfully shook hands with another node, it is able to exchange lightweight UDP-based messages. In the handshake procedure both nodes exchanged a 128-bit AES secret key which can now be used to encrypt the DHT-messages.

The `Messenger` implements two functionalities: A *UDP server* that listens for incoming UDP packets, as well as a *client socket* that sends out packets. Similar to the handshake server, the UDP message server also uses thread pools and worker threads to handle many requests concurrently, but adds a `BlockingQueue`-based message queue for incoming UDP packets. Besides receiving and sending messages, the `Messenger` also implements response notification for sent requests, and makes it possible to schedule timeouts if a response does not arrive within a specific time-frame.

Message Structure: `Message` and `EncryptedMessage`

In order to identify an arriving message or to create new packets, every exchanged UDP packet uses a strictly defined message format. The abstract base class, `Message`, only defines an *ID* and a *message type*, but passes on these properties to the classes `Request` and `Response`. Each Kademlia message inherits from these two abstract classes. Thus, the FIND_NODE RPC, for example, implies two classes, `FindNodeRequest` and `FindNodeResponse`. Kademlia's other RPCs are analogue.

Since KadS enforces secure communication, every DHT message is encrypted using the negotiated session key. The `EncryptedMessage` class realizes this requirement by encrypting each `Message` object for a given `VerifiedNode`. That is, before a message is packed in a UDP packet, it is encrypted for a specific verified recipient node.

In order to allow the recipient to identify the sender, each `EncryptedMessage` object is equipped with a plaintext representation of its node ID. The data sent between nodes in a KadS network therefore always has the form *(bit[160] senderID, int payloadLength, byte[] payload)*.

Sending Messages

Sending DHT messages such as FIND_NODE or STORE is pretty straightforward: Unlike TCP, the lightweight UDP does not establish a connection to a remote machine

before sending data to it. Instead, UDP allows sending out datagram packets to any machine on the Internet without having to care about keeping up a connection or having to confirm that packets arrived. Once the local KadS node exchanged a session key with a remote node, it can send lightweight UDP packets via its `Messenger` (cmp. listing 5.4).

```
/* Find K closest nodes */
List<VerifiedNode> closestNodes = new FindNodeOperation(..).execute();

/* Send STORE requests to each of them */
for(VerifiedNode recipientNode : closestNodes) {
  StoreRequest storeRequest = new StoreRequest(key, value);
  ...
  messenger.addResponseListener(storeRequest.getId(), recipientNode, this);
  messenger.send(recipientNode, storeRequest);
}
```

Listing 5.4: The store operation locates the K closest nodes to the given key and sends a STORE request to each of them.

KadS operations are not aware of the fact that the communication is encrypted. In fact, they only know that they have to shake hands with a node first before they are able to send DHT-related messages. Hence, message encryption is completely transparent to them. Instead, they simply use the `Messenger` to send out RPCs without having to cope with how to encrypt a request. In order to be notified as soon as a response arrives (or times out), it furthermore allows registering `ResponseListener`s for any request. That way, operations can communicate with many network nodes asynchronously.

Receiving Messages

While sending DHT messages is relatively simple, processing incoming packets is a more complex task. Before the local KadS node can determine what type of message it received, it has to find the sender in the routing tree and decrypt it with their session key.

As figure 5.3 shows, each UDP packet traverses many steps before it reaches the destined operation: After receiving a packet, the messenger simply puts it in a `BlockingQueue` of UDP packets. One of worker threads takes it and assembles it to an `EncryptedMessage` object. Then, the sender node is looked up in the local `RoutingTree` and if it is found, the payload is decrypted using the formerly negotiated session key, and either a `Request`- or `Response`-object is created. Depending on the type of the message, the worker thread creates a `RequestHandler`, or calls the previously registered `ResponseListener` (if there is any).

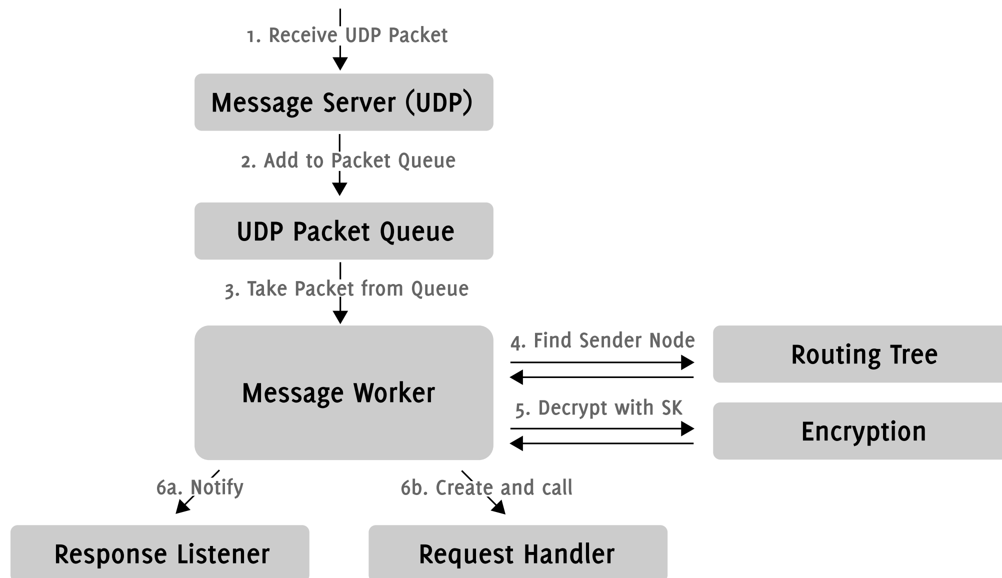


FIGURE 5.3: When the messenger receives a UDP packet (1), it first puts it into a `BlockingQueue` of UDP packets (2). One of the several worker threads takes it (3), creates an `EncryptedMessage` object and looks up the verified sender node in the local routing tree (4). If found, it uses the session key to decrypt the message to a `Response` or `Request` object (5). In the former case (6a), the worker passes the response to the waiting `ResponseListener`. In the latter case (6b), it creates a `RequestHandler` and calls it.

`RequestHandler`

If the local KadS node receives a `Request` from a verified node, the message worker thread uses the `RequestHandlerFactory` to create a `RequestHandler`.

For each Kademlia RPC, KadS provides a handler class to process incoming DHT requests. If, for instance, a `FindValueRequest` is received, the above factory creates a corresponding `FindValueRequestHandler`. This handler is then called by the message worker and responds to the sender node as soon as an answer is found. In this particular case, the handler would look up the requested key in its local hash table and respond accordingly. If the *(key, value)*-pair is found, it returns a `FindValueResponse`-object that contains the desired value. If, however, the local hash table does not contain the key, the node responds with the K closest nodes to the requested key.

5.1.6 Kademlia RPCs: `Operations`

Even though the above explanations describe a huge part of the system, they are not specifically related to the Kademlia protocol so far. In order to implement the protocol, the KadS system groups the major Kademlia functionalities in so called `Operations`.

Each of Kademlia's remote procedure calls defines specific requirements that each protocol-conform application must fulfill. According to the four RPCs, KadS concentrates the requirements in four corresponding `Operation` classes.

STORE-Operation

The STORE-RPC, for instance, is represented by the `StoreOperation`-class and simply implements Kademlia's requirements: Before the actual STORE-messages are sent to any node, Kademlia specifies that the K closest nodes to the given ID need to be found. As soon as they have been located, the given hash table entry will be stored at each of them. Therefore, this operation can be described very easily (cmp. listing 5.4).

FIND_NODE-Operation

In contrast to the simple STORE- and PING-operations, Kademlia's most important RPC, FIND_NODE, is much more complex and uses concurrency for enhanced performance. Since FIND_NODE and FIND_VALUE are pretty similar, the KadS implementation uses the abstract base class `FindOperation` to realize most of the functionality. The two child-classes `FindNodeOperation` and `FindValueOperation` alter only a few lines of code in order to implement the corresponding RPC.

As described in chapter 3.1.4, Kademlia's node lookup basically follows these steps:

1. Find the K closest nodes to the given identifier in the local routing tree.
2. Ask each of them if they know any closer nodes and wait for responses.
3. Every time a response arrives, check if it contains any closer, unasked nodes. If so, add them to a list and ask the closest node of this list.
4. If a node responds in time, update the last-seen date. If not, mark is as failed.
5. The operation finishes as soon as no closer, unasked nodes are returned.

In addition to Kademlia's requirements, KadS also needs to take into consideration that some of the returned nodes might be unknown to the local node, i.e. a handshake is necessary before communication with it.

5.1.7 Secure Communication and Encryption

The KadS core system implements certain functions to ensure secure communication between the trusted network nodes. While the actual operations and the Kademlia

messages are not aware of the encryption, the messaging and the handshake system take care of everything transparently.

The static `Encryption` class provides several functions to `encrypt/decrypt` byte-arrays using symmetric and asymmetric cryptography. In particular it implements two triple overloaded methods for encryption/decryption using a `SecretKey`, a `PrivateKey` or a `PublicKey`.

5.2 URL Blacklist

The URL blacklisting service uses the KadS infrastructure to securely store its data in an extendable, distributed network. Each blacklist node encapsulates a KadS node and enforces a specific data structure on the hash table entries. In order to communicate with clients, it furthermore provides a TCP/SSL server and a UDP-based query server. Like in the KadS network, every communication is encrypted.

In contrast to KadS, however, the URL blacklisting service does not implement any new technologies or revolutionary designs. Instead, it simply wraps around the KadS system and additionally equips each node with two client-accessible interfaces. To simplify the development of new clients and minimize the programming effort, the main focus of the outer interfaces is to use standard implementations and protocols. Instead of using a lightweight handshake protocol (like in KadS), the blacklisting service uses SSL to provide a secure socket, and uses the Advanced Encryption Standard (AES) for the further UDP communication.

5.2.1 Blacklist Node

As figure 5.1 already indicates, the blacklist service's main component is the `BlacklistNode`. It provides access to the data and is connected to the underlying KadS network. In order to get information about malicious Web sites, a client does not have to stick with a specific node, but can query any known blacklist node.

Each node contains four major components:

- `BlacklistServer`: The blacklist server is the first contact point for any client and implements a TCP/SSL server (by default bound to port 7001). Before any client can communicate with the service, it needs to establish a SSL connection with the server and exchange a symmetric session key. After successfully logging in, the

client can either write commands directly to the SSL socket, or use the session key to send queries to the UDP query server.

- `QueryServer`: The query server is a UDP-based server socket (also port 7001) that only supports querying the blacklist for specific domains. In order to send encrypted packets to it, each client must possess a session key from the initial handshake.

- Authenticated Users: Each blacklist node keeps track of the clients that signed on during the SSL handshake phase. Using a simple `Map` structure to store session identifiers and their corresponding `User` objects, the UDP query server is able to identify, decrypt and process incoming messages.

- `KadS` Node: As major part of each blacklist node, a `KadS` node provides access to the actual underlying distributed hash table. Every time a client likes to access domain-related information, the blacklist node calls the `get`-method of the KadS node to query the network.

When a blacklist node is started, it initializes the two servers and starts up its very own KadS node. In order to access the data of the DHT, the `BlacklistNode`'s internal `KadS` instance needs to connect to the network.

5.2.2 Client Classification

For clients, the only way to access the stored information is to get in contact with the servers of a blacklist node: While the `BlacklistServer` provides all functionality (such as query and store blacklist entries), the `QueryServer` only supports the query-command.

The URL blacklisting service knows two different types of clients.

- Untrustworthy Clients (Browser Plug-in): Normal clients do not need to authenticate with the blacklist server, but are completely anonymous entities. This thesis calls them *untrustworthy clients* and restricts their blacklist-access to *read-only*.

- Trustworthy Clients (Honeyclient): Unlike the untrustworthy clients, *trustworthy clients* have *read- and write-access* to the blacklisting service, i.e. they are able to store, update and delete entries from the blacklist. In order to prove their identity, they have to authenticate with a CA-signed public key certificate during the SSL handshake phase (*Client authentication*).

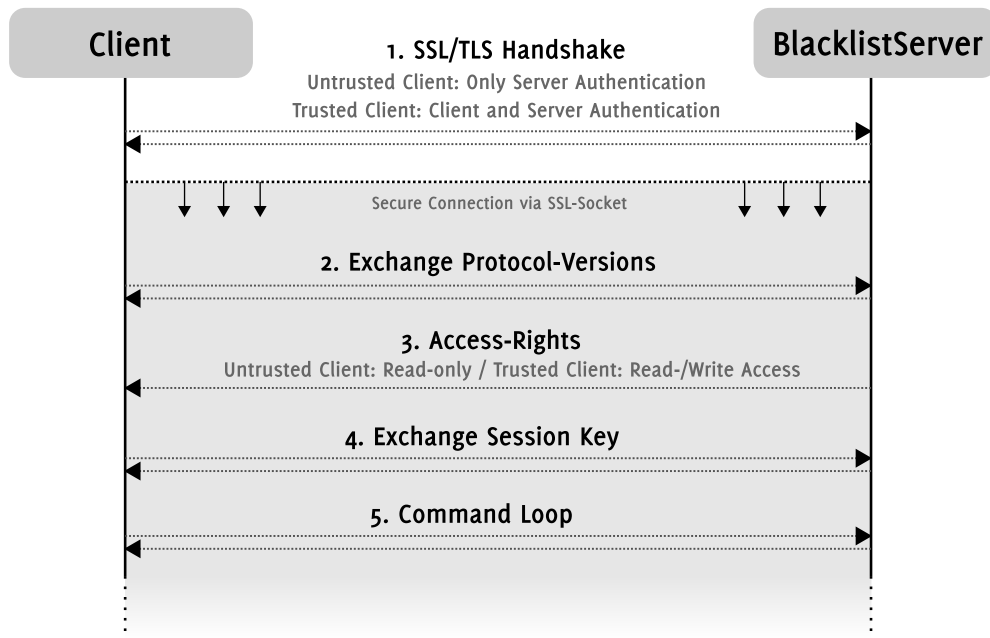


FIGURE 5.4: Each client needs to login at the `BlacklistServer` before it can send commands: While trustworthy clients need to authenticate with the server, untrustworthy clients can login anonymously. After exchanging a session key, `Query`s can either be send over the TCP/SSL socket, or to the UDP-based `QueryServer`.

5.2.3 Protocol: `BlacklistServer` and `QueryServer`

While their requirements and goals are different, the communication protocol of the two clients with the blacklisting service is pretty similar (cmp. figure 5.4):

Both start the communication by establishing a SSL connection (step 1) with the blacklist node's TCP server (`BlacklistServer`). While the untrustworthy client only verifies the server's identity (using the locally stored CA), the trustworthy client also has to prove its identity to the server. By sending its CA-signed certificate (cmp. step 7 of the SSL handshake, figure 3.5), the client is able to authenticate with the blacklist server.

Once the secure connection is established, both can communicate via the SSL socket. After exchanging protocol versions (step 2), the server notifies the client about the granted access rights (step 3): Untrustworthy clients always get read-only rights, while trustworthy clients get read- and write-privileges if their client authentication was successful.

In step 4, both client and server send 64-bits of random data in order to create a 128-bit AES secret-key (session key, SK). This key will only be needed if the client intends to communicate with the UDP-based query server.

After these four steps, the TCP/SSL server enters a command loop in which the client controls the actions. Because untrustworthy clients only have read-access, they can only

send the three commands NOP, QUERY and CLOSE. Trustworthy clients, however, are additionally allowed to perform STORE actions:

- NOP: In order to keep the TCP-socket alive, the client has to send *No Operation (NOP)* commands every 30 seconds.
- QUERY: To ask the service about specific domains, the client can send an encoded `Query` object the blacklist node. The server answers within a given time frame with an encoded `Entry` object that contains the blacklist results.
- STORE: By sending an `Entry`-object, trustworthy clients can store/overwrite entries. The blacklist server receives the entry and stores it in the KadS network.
- CLOSE: To close the TCP/SSL socket, the client has to send a CLOSE command.

`BlacklistServer`

The implementation of the `BlacklistServer` is very similar to the `HandshakeServer` of the KadS system: Both use thread pools and listen at a given TCP port.

Unlike the handshake server, however, the blacklist server binds a SSL socket and utilizes classes of the Java Secure Socket Extension (JSSE) [34]. Instead of simply creating an instance of Java's `ServerSocket` class, the blacklist node has to initialize a more complex `SSLContext` environment in order to create a `SSLServerSocket`.

Once this socket is created, it listens for new connections and follows the above described protocol. As soon as a client successfully performs all required steps, the blacklist server creates a new `User` object and saves it in the blacklist node's *authenticatedUsers* `Map`.

`QueryServer`

The `QueryServer` binds a UDP server socket and listens for incoming `Query` messages. Similar to the KadS node's message server, each UDP packet has to be encrypted using the previously negotiated session key.

Every time the server receives a UDP packet, it simply adds it to a `BlockingQueue`. One of the `QueryWorker`s takes the packet from the queue, looks up the session ID in the *authenticatedUsers*-`Map` and decrypts it with the secret key. It then calls the KadS node's `get`-method to look up the DHT-entry and returns an `Entry`-object to the client.

5.3 Examples

After briefly presenting the implementation of the KadS system and the URL blacklisting service, the following few lines of code will give an impression on how to use the system.

5.3.1 KadS Node

The core network consists of many communicating `KadS` nodes. Each computer can start as many nodes as its hardware allows, but has to provide a CA-signed certificate and a matching private key for each of them.

In order to adjust each node individually, KadS allows a detailed configuration for each instance. The `Config` class represents a config file in which all properties of a node can be adjusted (listing 5.5):

```
/* To prove other node's identity, each node must
   possess a copy of the network's CA certificate */
kads.ca              = nodes/ca.cer

/* Each node needs to provide a CA-signed keypair
   to define its local identity. */
kads.node.cer        = nodes/499/node.cer
kads.node.key        = nodes/499/node.key

/* In case a different TCP/UDP port shall be used for the
   handshake- and message server, one can override this property */
kads.port            = 6852

/* To analyze the traffic and/or queries, each node
   has its own log-file with adjustable log-level */
kads.logfile         = nodes/499/log
kads.loglevel        = debug

/* KadS nodes can automatically try to connect to other
   nodes using a given list of IPs. */
kads.boot.sleep      = 2000-10000
kads.boot.nodelist   = nodes/bootstrap.list
kads.boot.tries      = 5
```

Listing 5.5: Each KadS node allows detailed configuration.

After configuring, one can start the KadS node using the config file. After a short initialization and connecting to an existing KadS network, the DHT can be used like other

hash tables. As listing 5.6 shows, one can put any `Serializable` object in the DHT and retrieve entries exactly like with normal Java `Maps`:

```
/* Parse the config file and load the CA certificate,
   and the local node's key-pair. */
Config config = new Config(new File("nodes/499"));

/* Initialize the KadS node and start handshake-
   and message server */
KadS kads = new KadS(config);

/* Connect to an existing KadS network */
kads.connect(new InetSocketAddress("node1.kads.dyndns.org", 6852));

/* Store an object in the DHT */
kads.put("cities", new String[] { "Mannheim", "Heidelberg" });

/* Retrieve an object from the DHT */
List<Country> countries = (List<Country>) kads.get("countries");

/* Disconnect this KadS node */
kads.close();
```

Listing 5.6: After initializing the KadS node and connecting to an existing network the DHT can be used like any other hash table.

5.3.2 Blacklist Node and Clients

Each `BlacklistNode` is wrapped around a KadS node and carries an instance of the `KadS` class with it. Therefore, instantiating a blacklist node is very similar to the KadS node (cmp. listing 5.6).

Connecting to a blacklist node as a client, however, requires a different proceeding: While any `UntrustworthyClient` only must provide a copy of the blacklist network's CA certificate, the `TrustworthyClients` also need a CA-signed certificate and a matching private key (similar to a KadS node).

Trustworthy Clients

The following listing shows how a honeyclient could use a `TrustworthyClient` object to store blacklist entries in the DHT:

```
TrustworthyClient client =
  new TrustworthyClient("Honey1.cer", "Honey1.key", "NetworkCA.cer");
```

```
/* Connect to a blacklist node in the network */
client.connect(new InetSocketAddress("node1.blacklist.dyndns.org", 7001));

if (!client.getAccessRights().hasWriteAccess())
  throw new Exception("No write access!");

/* Store a blacklist entry (via TCP/SSL socket) */
client.store(new Entry("uni-mannheim.de", Entry.CLEAN));
client.store(new Entry("uni-heidelberg.de", Entry.INFECTED));
...

/* After all entries are stored, the TCP/SSL socket can be closed */
client.disconnect();
```

Listing 5.7: Trustworthy clients need to provide a CA-signed certificate and a corresponding private key to alter data in the blacklist.

Untrustworthy Clients

While the trustworthy clients are intended to be used by the honeynet or other trusted entities, the untrustworthy clients have been designed to work as plug-in for a browser. Listing 5.8 shows a Java-like pseudo code of how the usage is intended:

```
public void onBrowserStart() {
  blacklistClient = new UntrustworthyClient("NetworkCA.cer");

  /* Connect to a blacklist node in the network */
  blacklistClient.connect(
    new InetSocketAddress("node1.blacklist.dyndns.org", 7001) );
}

public void onBeforeOpenWebsite(String domain, String path, ...) {
  Entry entry = blacklistClient.query(domain);

  if (entry.getAnalysisCode() == Entry.INFECTED)
    throw new MaliciousWebsiteException(...);
  ...
}
```

Listing 5.8: Untrustworthy clients only connect one to the blacklist node to exchange a session key. Once the initial handshake is done they communicate via the lightweight UDP.

When the browser starts, the untrustworthy client connects to an arbitrary blacklist node and exchanges a session key with it. Once this is done, it hooks in the browser

event `onBeforeOpenWebsite` (or similar) and queries the blacklist node. Depending on the result, it either allows the access or denies it.

Chapter 6

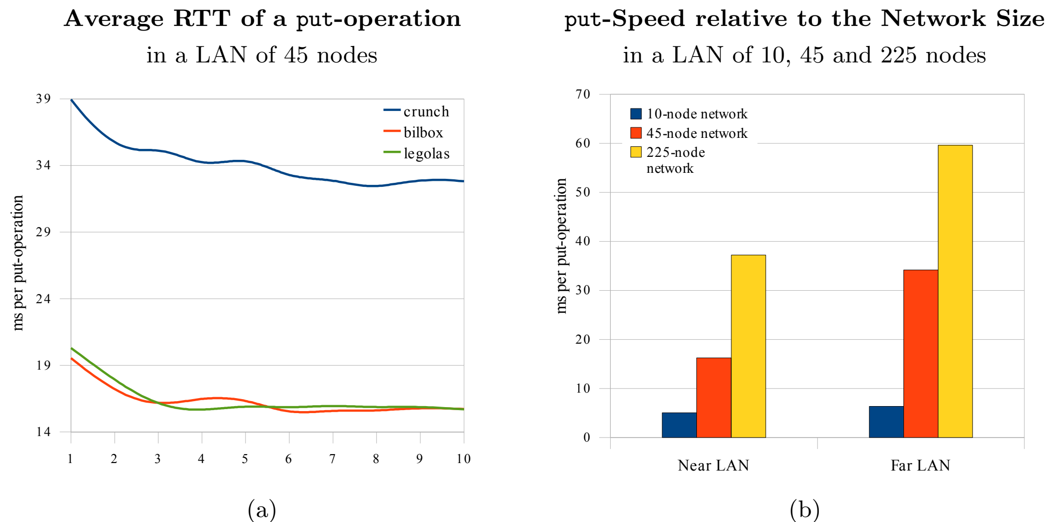
Testing

Even though the application was designed to work in a large network environment, one can hardly eliminate all possible design and implementation errors. In order to briefly evaluate the quality of the application, this chapter focuses on analyzing the road capability of the implementation. In particular, it sets up a test environment of a few hundred nodes and executes several performance tests on both systems, the core network and the blacklisting service.

6.1 Setup

The following tests all take place in a local area network (LAN) with 45 PCs. In particular, they use the infrastructure of the computer pool of the Faculty of Mathematics and Computer Science at the University of Mannheim (π-pool).

Each pool PC is a standard desktop PC, i.e. a Intel Core2 Duo 2x2GHz with 4 GB RAM. Every PC serves as host for one or many blacklist nodes and can be controlled via several bash-scripts.



(a) (b)

FIGURE 6.1: (a) The average roundtrip-times of KadS operations decrease as nodes get to know each other. While the operation in a far LAN (crunch) takes longer, one `put`-call on a KadS DHT takes about 16ms in near LAN environment. (b) Since nodes have to shake hands before they exchange DHT-related messages, the RTT of KadS methods increases faster than Kademlia's RPCs.

6.2 Average RTT of KadS' put-Operation

In order to evaluate the operation speed for the DHT-methods, this test focuses on the determination of the average request/response round-trip times (RTT) from inside and outside the network.

Since a DHT can consist of only a few nodes as well as of several thousand nodes, this test tries to take the network size into consideration. By performing it on different network sizes, the results can be evaluated in relation to the number of participating nodes.

In representation of all KadS methods, the test measures the average RTT for storing values in the network, i.e. the speed of the `put`-operation. By repeatedly storing 1000 randomly generated entries in the DHT, one can determine the average RTT:

```
long timeBeforeLoop = System.currentTimeMillis();

for (int i=0; i<1000; i++)
  kads.put(randomKeys[i], randomValues[i]);

System.out.println(
  ((double)(System.currentTimeMillis()-timeBeforeLoop)/1000) + "ms");
```

LISTING 6.1: Each iteration of this test stores 1000 randomly generated key/value-pairs in the distributed hash table.

In a real world scenario, the DHT (or the blacklist) might be filled from outside the network, or from further away in the local area network.[1] For that reason, this test is executed from three different machines: While *legolas* and *bilbox* are located in the near LAN and are part of the DHT, *crunch* is located further away in the network.

Results

The test has been performed on a 10-, 45- and 225-node network. All tests returned significantly different results and showed the immediate dependence of the network size to the operation speed.

As figure 6.1b shows, the operation time increases from an average of 5ms in the small network, to 37ms in a larger network. While the results of the farther machine (*crunch*) also include the network latency times, the measured times of the other two nodes represent the immediate operation time (with negligible latency values).

While the average RTT of Kademlia's operations grows logarithmically to the number of nodes [13], the KadS network cannot offer this property: The secure nature of KadS prevents uncomplicated stranger-to-stranger communication by enforcing a handshake preliminary to every message exchange. The initial negotiations take a lot of time as nodes have to perform several handshakes for each `get-`/`put`-operation.

Figure 6.1a makes this peculiarity of the KadS network visible: While the first few thousand requests take between 17ms and 20ms, the same operation only needs 15ms when more nodes know eachother. The fewer handshakes a node has to perform during a lookup operation, the faster does it get the results. In fact, in small networks, it is even possible to reach Kademlia's speed: Once all nodes shook hands with each other, they are able to exchange messages with very little delays.

[1]This consideration is directly related to the later scope the URL blacklisting application: The KadS network is meant to be fed by the Honeynet, which is located in the LAN, but further away.

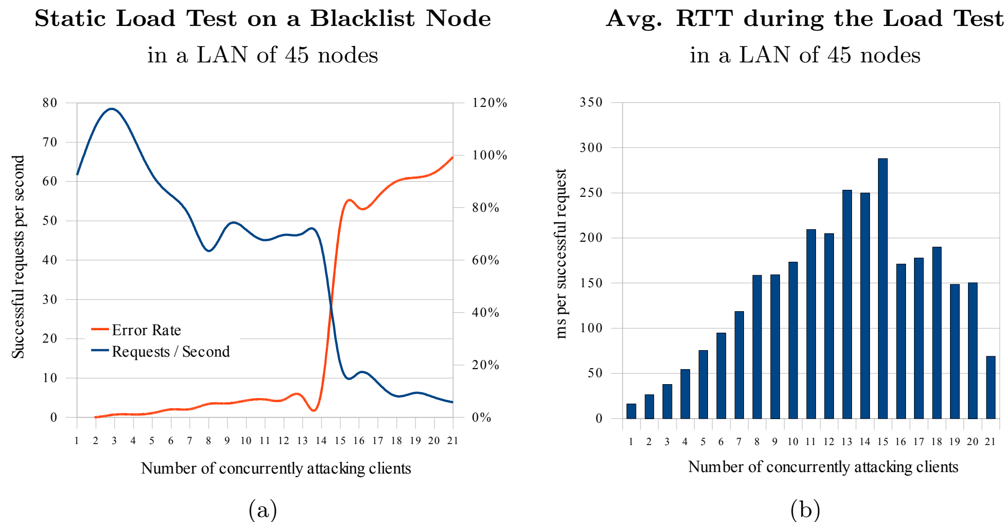


FIGURE 6.2: (a) A single blacklist node can handle up to 79 requests/second as long as it does not have to handle too many UDP packets concurrently. The more packets it receives at the same time, the higher is the error rate. When the level of available worker threads is reached, the error rate rises dramatically and the number of successful requests drops to nearly zero. (b) The more concurrent UDP packets arrive, the more increases the average RTT. As soon as no more worker threads are available, most packets are denied and the few processed packets have a normal RTT.

6.3 Static Load Test on a Blacklist Node

In order to determine how many requests per minute each blacklist node is able to handle, the second test simulates many browser clients. By querying the node several thousand times with multiple client instances, one can figure out the maximum number of processable requests for a single blacklist node.

Assumptions

Like every server-system, both blacklist service and the underlying KadS core network are limited in terms of processable concurrent clients. However, they provide parameters to adjust the concurrency level and the number of available worker threads.

This thesis assumes that no more than 15 requests have to be handled concurrently. Even though this value is lower than the default values in some well-known servers,[2] it is appropriate to the processor power of the pool PCs. However, due to the fact that both UDP-based servers (KadS' `Messenger` and the blacklist's `QueryServer`) use a FIFO queue to handle requests, the number of processable requests is slightly higher than 15.

[2]The Apache Web server predefines 25-75 worker threads, the MySQL database server only 10.

Test

The test simulates several untrustworthy blacklist clients that *concurrently* send UDP packets to a single blacklist node. Each client repeatedly queries the same node for 1 minute and counts the number of successful and erroneous requests. Before a client considers a request as failed, it waits 5 seconds for its response.

In a real world scenario, each *untrustworthy client* represents a plug-in in the user's browser: Before surfing to a Web page, the browser plug-in queries the blacklist node and asks for the malignity-status of the site. After the user received the response, he or she visits the Web page and it takes a *couple of seconds* until a new site is opened. While the user is browsing, the blacklist node can answer requests from other clients.

In this test, however, the clients do not simulate the idle time between their queries. Instead, they *constantly* send new requests to the node to get an estimate of the maximum number of processable queries.

In each iteration, the test increases the number of concurrently penetrating clients until the load reaches a critical point and the node denies the service.

Results

The most obvious result of this test is the fact that the more concurrent packets the blacklist node has to handle, the higher is the RTT for each request (fig. 6.2b). In fact, the average time per request constantly increases from 16ms to almost 300ms in the first phase of the test.

However, as soon as the level of 15 concurrently penetrating clients is reached, the RTT drops significantly: Due to the fact that only 15 worker threads are available to handle the requests, the blacklist node simply cannot process most of the packets. Therefore, when more than 15 clients are constantly sending packets to the node, it simply denies the service for most packets and processes only very few of the arriving requests. Because these few packets can be handled at normal speed, the RTT of the successful requests decreases in the second phase of the test.

Not only the RTT increases with the number of penetrating clients, but also the error rate constantly rises during the test (fig. 6.2a). While less than 5% of the requests fail when the blacklist node handles up to 7 attacking clients at the same time, and only up to 10% for 14 clients, the error rate drastically rises to 74% when the level of 15 attackers is reached.

At the same time, the number of successful requests per second drops from its peak value of 78 to only 13 requests/second for 15 concurrent attackers (fig. 6.2a).

Chapter 7

Conclusion

Peer-to-peer systems have reached great publicity in the last few years, but most systems still lack of authentication mechanisms or communication encryption. While classical client-server systems are mostly equipped with numerous security features and implement PKI support through TLS/SSL, almost all P2P networks are open for everyone.

In this thesis, the well-known P2P-based protocol Kademlia has been extended to an access-restricted secure distributed hash table: The new PKI-supported DHT protocol *KadS* implements certificate-based authentication and encrypts the communication between participating nodes. While traditional P2P networks allow every node to join and communicate with one another, KadS nodes have to possess a valid CA-signed public key certificate and a matching private key to join the network. Therefore, KadS allows the creation of a trustworthy P2P network and can be used to store confidential information.

As primary application for the new KadS network, the thesis implemented a P2P-based URL blacklisting service on-top of the secure DHT: Similar to other solutions, the developed blacklist stores information about malicious Web sites and provides an interface for end-users to access it. However, unlike most services, the underlying database is based on a distributed infrastructure rather than a simple server system.

Starting with chapter 2, the thesis introduced two existing blacklisting services to give a deeper insight of the subject and explain the core technologies of their implementations. While both Google Safe Browsing and Microsoft SmartScreen work on a large scale, they use a traditional client-server architecture as underlying system. The approach of this thesis, however, uses P2P to store the data in a distributed manner and the concepts of a PKI to restrict the access and encrypt the communication. For that matter, chapter 3 explained the fundamental elements of this thesis' central topics: While the first part

described the peer-to-peer paradigm and introduced the concepts of distributed hash tables, the second part focused on explaining the basics of public key infrastructures and their entities.

Having explicated the required concepts of P2P and PKI, chapter 4 described the scope of the thesis' application and defined its major goals: While the first goal targeted on making the honeynet-provided data publicly available for a broad audience through a URL blacklisting service, the second goal decided to use P2P as the underlying technology. Unlike client-server architectures, P2P-based systems do not have to use additional technologies to expand the service on a greater scale, but offer scalability and fault-tolerance *by design*. For the implementation, the discussion chose Kademlia as foundation for the construction of a new PKI-supported DHT protocol. Because of its concurrent nature and the superior routing algorithm, it has been determined as the most suitable protocol to embed PKI technologies.

After discussing possible application designs and having illustrated the chosen approach, chapter 5 introduced the final implementation of the KadS network and the blacklisting service. By demonstrating the most important components of the system, the chapter presented the prototype of the KadS protocol and the on-top service in great detail. In particular, it showed how the individual parts work together and how messages flow within the system.

Having explained the implementation and given a few examples of how to use both systems, chapter 6 finally evaluated the outcome of the thesis' work: Using a few of the university's computers, it simulated different network sizes and performed two tests on the system. The first test measured the average round-trip time of KadS' operations in relation to the network size and determined that even though KadS has been designed to scale very easily, its secure design causes the speed to reduce significantly as the number of network nodes increases. That is, unlike the original Kademlia design, its performance strongly depends on how many nodes participate in the network and thereby failed one of the original requirements. However, for its intended purpose as URL blacklisting service, only very few computers are required so that the decreasing performance carries no weight.

The second test measured how much load a single blacklist node can handle without having to deny the service for most requests. As the results show, as long as not too many clients constantly query a single node, the blacklist (and the underlying KadS system) can handle up to 4700 requests per minute with an error rate lower than 2%. Given that the network can consist of many nodes and clients can choose the node they like to query, the system can resist at least small-sized attacks.

Although the thesis' work was highly prototypical, the outcome reached most of its previously defined goals. In fact, it created a PKI-supported distributed infrastructure for storing large amounts of arbitrary data, it allows concurrent access and is somewhat resistant against attacks. At the same time, the system encrypts every connection and only lets trusted entities access the network.

Even though the final result does not fulfill all initial requirements (such as scalability), the overall outcome is satisfactory and enables the possibility of future research topics.

Bibliography

[1] Internet Crime Complaint Center. IC3 2008 Annual Report on Internet Crime. 2009. URL http://www.ic3.gov/media/2009/090331.aspx.

[2] Kaspersky Lab. Kaspersky Security Bulletin: Statistics 2008. March 2009. URL http://www.viruslist.com/en/analysis?pubid=204792052.

[3] Kaspersky Lab. Kaspersky Security Bulletin: Malware evolution 2008. March 2009. URL http://www.viruslist.com/en/analysis?pubid=204792051.

[4] Glen Murphy. New Firefox extensions. 2005. URL http://googleblog.blogspot.com/2005/12/new-firefox-extensions.html.

[5] Mozilla Foundation. Firefox Phishing and Malware Protection. 2009. URL http://www.mozilla.com/en-US/firefox/phishing-protection/.

[6] Apple Inc. Software License Agreement for Safari. 2009. URL http://images.apple.com/legal/sla/docs/SafariMac.pdf.

[7] ZDNet. Study: IE8's SmartScreen leads in malware protection. 2009. URL http://blogs.zdnet.com/security/?p=2981.

[8] Mozilla Foundation. Phishing Protection: Design Documentation. 2009. URL http://code.google.com/p/google-safe-browsing/wiki/Protocolv2Spec.

[9] Mozilla Foundation. Phishing Protection: Design Documentation. 2009. URL https://wiki.mozilla.org/Phishing_Protection:_Design_Documentation.

[10] MacWorld.com MacJournals.com. Inside Safari 3.2's anti-phishing features. 2008. URL http://www.macworld.com/article/137094/2008/11/safari_safe_browsing.html.

[11] Microsoft Corporation. Principles behind IE7s Phishing Filter. 2005. URL http://blogs.msdn.com/ie/archive/2005/08/31/458663.aspx.

[12] Microsoft Corporation. IE8 Security Part VIII: SmartScreen Filter Release Candidate Update. 2009. URL http://blogs.msdn.com/ie/archive/2009/02/09/

ie8-security-part-viii-smartscreen-filter-release-candidate-update.aspx.

[13] Ralf Steinmetz. *Peer-to-peer systems and applications*. 2005.

[14] A. Oram. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. 2001.

[15] H. Garcia-Molina B. Yang. Comparing Hybrid Peer-to-Peer Systems. 2001. URL www.dia.uniroma3.it/~vldbproc/060_561.pdf.

[16] K. Börner G. Fletcher, H. Sheth. Unstructured Peer-to-Peer Networks: Topological Properties and Search Performance. 2005.

[17] Robert Morris Ion Stoica. Chord: Ascalable Peer-To-Peer lookup service for internet applications. 2001. URL http://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf.

[18] P. Druschel A. Rowstron. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. 2001. URL http://research.microsoft.com/~antr/PAST/pastry.pdf.

[19] Paul Francis Sylvia Ratnasamy. A scalable content addressable network. 2001. URL http://www.cs.cornell.edu/people/francis/p13-ratnasamy.pdf.

[20] John Kubiatowicz Ben Y. Zhao. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. 2001. URL http://cs-www.cs.yale.edu/homes/arvind/cs425/doc/tapestry.pdf.

[21] E. Lehman D. Karger. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. 1997. URL http://tinyurl.com/akamai-com.

[22] Diomidis Spinellis Stephanos Androutsellis-Theotokis. A survey of peer-to-peer content distribution technologies. 2004. URL http://doi.acm.org/10.1145/1041680.1041681.

[23] Andrea W. Richa C. Greg Plaxton, Rajmohan Rajaraman. Accessing nearby copies of replicated objects in a distributed environment. 1997.

[24] David Mazières Petar Maymounkov. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. 2002. URL http://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf.

[25] John R. Douceur. Accessing nearby copies of replicated objects in a distributed environment. 2002. URL http://www.cs.rice.edu/Conferences/IPTPS02/101.pdf.

[26] Steve Lloyd Carlisle Adams. *Understanding PKI*. 2007.

[27] Martin Hellman Witfield Diffie. New directions in cryptography. 1976.

[28] Internet Engineering Task Force. RFC 5280 - Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. URL `http://tools.ietf.org/html/rfc5280#section-3.1`.

[29] International Telecommunication Union (ITU). About ITU. 2009. URL `http://www.itu.int/net/about/index.aspx`.

[30] ITU-T Recommendation X.509. Directory: Authetication Framework. 1997. URL `http://www.itu.int/rec/T-REC-X.509-199708-S/e`.

[31] ITU-T Recommendation. Directory: Public Key and Attribute Certificate Frameworks. 2000. URL `http://www.itu.int/rec/T-REC-X.509`.

[32] ITU-T Recommendation X.509. Directory: Overview of concepts, models and services. 1988. URL `http://www.itu.int/rec/T-REC-X.500-198811-S/en`.

[33] Eric Rescorla. *SSL and TLS*. 2001.

[34] Sun Microsystems Inc. JSSE Reference Guide for Java SE 6. 2006. URL `http://java.sun.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html`.

[35] Laboratory of Dependable Distributed Systems University of Mannheim. German Honeynet Project - Honeynet research. 2009. URL `http://pi1.informatik.uni-mannheim.de/index.php?pagecontent=site/Research.menu/Honeynet.page`.

[36] Sun Microsystems Inc. Java Cryptography Architecture. 2007. URL `http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html`.