Università degli Studi di Padova
Departimento di Ingegneria dell'Informazione

TESI DI DOTTORATO

Randomness, Age, Work: Ingredients for Secure Distributed Hash Tables

Dottorando:
Vincenzo-Maria Cappelleri

Supervisore:
Prof. Enoch Peserico

Direttore della Scuola:
Prof. Matteo Bertocco

Coordinatore d'indirizzo:
Prof. Carlo Ferrari

January 31st, 2017

Preface

My journey among distributed systems began when I was an undergraduate student. Since then I became immensely fascinated from their aesthetically pleasing complexity, especially when dealing with spontaneous networks. The enormous number of things that can go wrong, the variety of the machines hosting their instances, it all reminds me of the complexity in human society.

The security problems affecting these contraptions however has always deeply bugged me. With this work, even in its simplicity, I hope to offer the community yet another idea on how to put a patch on it.

Stylistically, this dissertation is written using the plural noun. In "The Sybil Attack" Douceur wrote: "Use of the plural pronoun is customary even in solely authored research papers; however, given the subject of the present paper, its use herein is particularly ironic". For the importance that his work had on all my endeavor, I thought it was appropriate to join in this little joke.

I've never been good at inscriptions. Nonetheless my gratitude goes to friends and family and, albeit I will not explicitly write their names, they definitely know my thoughts are for them as I write these few lines.

Vincenzo-Maria Cappelleri
Padova
January 31st, 2017

Abstract

Distributed Hash Tables (DHTs) are a popular and natural choice when dealing with dynamic resource location and routing. DHTs basically provide two main functions: saving (key, value) records in a network environment and, given a key, find the node responsible for it, optionally retrieving the associated value. However, all predominant DHT designs suffer a number of security flaws that expose nodes and stored data to a number of malicious attacks, ranging from disrupting correct DHT routing to corrupting data or making it unavailable. Thus even if DHTs are a standard layer for some mainstream systems (like BitTorrent or KAD clients), said vulnerabilities may prevent more security-aware systems from taking advantage of the ease of indexing and publishing on DHTs.

Through the years a variety of solutions to the security flaws of DHTs have been proposed both from academia and practitioners, ranging from authentication via Central Authorities to social-network based ones. These solutions are often tailored to DHT specific implementations, simply try to mitigate without eliminating hostile actions aimed at resources or nodes. Moreover all these solutions often sports serious limitations or make strong assumptions on the underlying network.

We present, after after providing a useful abstract model of the DHT protocol and infrastructure, two new primitives. We extend a "standard" proof-of-work primitive making of it also a "proof of age" primitive (informally, allowing a node to prove it is "sufficiently old") and a "shared random seed" primitive (informally, producing a new, shared, seed that was completely unpredictable in a "sufficiently remote" past). These primitives are then integrated into the basic DHT model obtaining an "enhanced" DHT design, resilient to many common attacks. This work also shows how to adapt a Block Chain scheme – a continuously growing list of records (or *blocks*) protected from alteration or forgery – to provide a possible infrastructure for our proposed secure design. Finally a working proof-of-concept software implementing an "enhanced" Kademlia-based DHT is presented, together with some experimental results showing that, in practice,

the performance overhead of the additional security layer is more than tolerable.

Therefore this work provides a threefold contribution. It describes a general set of new primitives (adaptable to any DHT matching our basic model) achieving a secure DHT; it proposes an actionable design to attain said primitives; it makes public a proof-of-concept implementation of a full "enhanced" DHT system, which a preliminary performance evaluation shows to be actually usable in practice.

This work was supported by a national grant ("legge 170") on "innovative broadband telecommunication systems, possibly with satellite usage, for different use cases regarding security, prevention and intervention in case of natural disasters".

Abstract

Nel contesto dell'indirizzamento dinamico basato su risorse le *Tabelle di Hash Distribuite* (DHT) si rivelano una scelta naturale oltre che molto apprezzata. Le DHT forniscono due funzioni principali: il salvataggio di coppie (chiave, valore) e, data una chiave, la localizzazione del nodo per essa responsabile, opzionalmente unita al recupero del valore associato. La maggior parte delle DHT realizzate sono ad ogni modo vulnerabili a falle di sicurezza che espongono i nodi ed i dati salvati ad un certo numero di possibili attacchi. Tali attacchi spaziano dall'impedire il corretto instradamento sulla DHT al corrompere o rendere indisponibili i dati. Anche se le DHT sono uno standard *de facto* in sistemi molto diffusi (come per esempio i client di BitTorrent o per la rete KAD) la debolezza di fronte a questi attacchi potrebbe tuttavia impedirne l'adozione da parte di sistemi maggiormente incentrati sulla sicurezza, pur potendo trarre vantaggio dalla facilità di indicizzazione e pubblicazione delle DHT.

Nel corso degli anni, sia da parte della comunità accademica che da parte di sviluppatori professionisti, sono state proposte molte possibili soluzioni al problema di sicurezza della DHT, spaziando da idee basate sul controllo esercitato da parte di Autorità Centrali a meccanismi basati sulle social network. Le proposte sono spesso personalizzate per specifiche realizzazioni delle DHT o, spesso, cercano semplicemente di mitigare senza eliminare la possibilità di azioni ostili verso i nodi o le risorse. Inoltre le soluzioni proposte spesso dimostrano di essere seriamente limitate o basate su assunzioni piuttosto forti relativamente alla rete di riferimento.

In questo lavoro, dopo aver fornito un'utile e generica astrazione del protocollo e delle infrastrutture di una DHT, presentiamo due nuove primitive. Estendiamo la "normale" funzione di proof-of-work facendo si che offra anche una "prova d'età" (ossia, informalmente, permette di provare che un nodo sia sufficientemente "anziano") ed una primitiva che permetta l'accesso ad un seme randomico distribuito. Utilizzando queste due nuove primitive ed integrandole nell'astrazione basilare otteniamo una DHT "migliorata", resistente a molti degi comuni attacchi inferti a questi

sistemi. Inoltre mostreremo come un sistema basato sulle Block Chain – una collezione di "blocchi di dati" protetta contro la contraffazione – possa fornire una possibile fondazione per la nostra DHT migliorata. Infine abbiamo realizzato un software prototipo che realizza una DHT sicura basata sul sistema Kademlia. Utilizzando questo software abbiamo condotto degli esperimenti, dimostrando come questo sistema sia utilizzabile in pratica nonostante il lavoro addizionale richiesto dai nodi.

Concludendo questo lavoro forniamo il seguente contributo: descriviamo un nuovo insieme di primitive per ottenere una DHT sicura (adattabile ad ogni sistema conforme alla nostra definizione di DHT), proponiamo un'architettura concreta per ottenere una DHT migliorata, ed annunciamo una versione prototipale e funzionante di questo sistema.

Questo lavoro è stato possibile tramite una borsa di studio ad ambito vincolato ("legge 170") sul tema "sistemi di telecomunicazione innovativi a larga banda anche con impiego di satelliti per utenze differenziate in materia di sicurezza, prevenzione e intervento in caso di catastrofi naturali"

Contents

List of Figures viii

List of Tables xii

List of Acronyms xv

1 Introduction 1

2 The Distributed Hash Table 9
- 2.1 Description . . . 9
- 2.2 A basic DHT model . . . 11
- 2.3 DHT examples . . . 14
  - 2.3.1 CAN . . . 15
  - 2.3.2 Chord . . . 17
  - 2.3.3 Pastry . . . 19
  - 2.3.4 Tapestry . . . 20
  - 2.3.5 Kademlia . . . 22

3 DHT Security 27
- 3.1 Sybil attack . . . 28
  - 3.1.1 At a glance . . . 28
  - 3.1.2 Practical relevance . . . 29
  - 3.1.3 Proposed solutions . . . 31
  - 3.1.4 Open issues . . . 32
- 3.2 Node insertion . . . 33
  - 3.2.1 At a glance . . . 33
  - 3.2.2 Practical relevance . . . 34
  - 3.2.3 Proposed solutions . . . 35
  - 3.2.4 Open issues . . . 36
- 3.3 Publish attack . . . 37

3.3.1 At a glance . . . 37
3.3.2 Practical relevance . . . 38
3.3.3 Proposed solutions . . . 39
3.3.4 Open issues . . . 40
3.4 Eclipse attack . . . 41
3.4.1 At a glance . . . 41
3.4.2 Practical relevance . . . 42
3.4.3 Proposed solutions . . . 43
3.4.4 Open issues . . . 44

4 Our new secure DHT 45

5 Security guarantees 51
5.1 Preliminaries . . . 51
5.2 Assumptions . . . 52
5.3 Proximity guarantees . . . 53
5.4 Poisoning the routing tables . . . 54
5.5 Summary of changes . . . 56

6 New DHT primitives through a Block Chain 57
6.1 A generic block chain model . . . 57
6.2 A Block Chain enabling the new primitives . . . 64
6.3 Current block chains . . . 67
6.3.1 BitCoin, the first block chain . . . 67
6.3.2 Other notable block chains . . . 71
6.3.3 NameCoin . . . 73

7 A secure DHT implementation 75
7.1 An easy-to-use secure DHT . . . 75
7.2 System architecture . . . 76
7.2.1 The Block Chain module . . . 79
7.2.2 The DHT module . . . 81
7.3 Future work . . . 85

8 Experimental evaluation 89
8.1 Experimental set up . . . 90
8.2 Performance evaluation . . . 94
8.2.1 Block chain related tests . . . 94
8.2.2 DHT tests . . . 97

9 Conclusions 111

Bibliography 115

List of Figures

2.1 Example of a CAN 2-d virtual space partitioned among five nodes. 15

2.2 Example of a 16 nodes Chord network. One of the nodes shows its *fingers* and its *predecessor pointer*. Solid arrows (inside the *identifier circle*) represent *fingers* to the other nodes, while the dotted arrow (outside of the *identifier circle*) represents the *predecessor pointer*. . . 18

2.3 Example of a Kademlia network using 3-bits node IDs. All nodes from each of the three highlighted areas will be referenced in the same k-bucket in the selected node's (with ID 101) routing table. 23

3.1 A *Sybil attack* example. On a network where honest nodes only deploy one ID each, an attacker (the one marked with a star) deploys several IDs (marked in red). 29

3.2 A *Node insertion attack* example. In the left part of the image a Chord-like network is shown (see Section 2.3.2 for a reference on Chord mechanics). The placeholder marked with an @ symbol represent the target resource's ID position. The rightful node responsible for the resource is one the blue one, marked with a small check sign upon it. In the right part of the image is depicted the same network after a malicious node (the red one) perform a *Node insertion attack*, purposely becoming the closest node to target resource. All *look-up* queries for the resources will be directed toward the attacker node after its inception. 34

3.3 An *Eclipse attack* example. The right part of the image shows a generic *Distributed Hash Table* (DHT) network. The node on the extreme left part of the image wanted to join the DHT but it ended up connecting only to four malicious nodes (marked in red). Thus all victim node's connections to the main DHT network are carried out throughout the set of malicious nodes that are able, if and when they desire, to isolate the victim from the rest of the network. 41

6.1 A graphical representation of a single block chain *block* and all its basic components. 58

6.2 An example of a binary Merkle Tree built on *transactions*. The root node is a convenient digest, representing the whole *transactions* collection's *hash* in the *block header*. 59

6.3 An example of three *blocks* from an hypothetical block chain. The link between two subsequent *blocks*, consisting on the predecessor's *header*'s *hash*, is represented by the arrows. 60

6.4 A simplified transaction verification is shown in this picture. It is possible to check if a *transaction* is contained in a collection without knowing all other *transactions* if the collection itself is organized as a binary Merkle Tree, just by checking the tree's root *hash*. In this example to check if a certain *transaction* (marked with the star and blue colored) is contained in a collection whose Merkle Tree *hash* is known just three more *hash* values are required to compute the root value: they are shown as the red *hashes*. In the image the whole Merkle Tree is completed (with light gray elements and dotted arrows) for comparison. 62

8.1 Blocks' computation times during block chain test. The red vertical dashed line marks the point after which no new miner nodes were spawned in the network, leaving it in a steady computation regime. The green dotted horizontal line shows the average block's computation time during the second half of the experiment (after block 200). 95

8.2 Plot of how the hash value of key string "Lorem ipsum dolor sit amet" was changed by random seeds during the performance tests. 96

8.3 Pictorial representation of how the hash value of key string "Lorem ipsum dolor sit amet" was changed by random seeds ranging from seed number 12 to seed number 17 during the performance tests. 96

8.4 Store times recorded on a *Random-Age-Work DHT* (RAW DHT) node running a block chain module *thick* node. Both plots show the linear and logarithmic fit to the data. (a) only shows mean values, while (b) shows also all the sampled values. 98

8.5 Store times recorded on a RAW DHT node running a block chain module *thin* node. Both plots show the linear and logarithmic fit to the data. (a) only shows mean values, while (b) shows also all the sampled values. 99

8.6 Estimation of store times on a RAW DHT network sizing up to 1 million nodes. The dashed and dot-dashed lines are the plots of logarithmic fits shown in Figure 8.4 and Figure 8.5. The solid line instead, is the plot of the OLS logarithmic fit on the whole body of data. 100

8.7 Look-up times recorded on a RAW DHT node running a block chain module *thick* node. Both plots show the linear and logarithmic fit to the data. (a) only shows mean values, while (b) shows also all the sampled values. 102

8.8 Look-up times recorded on a RAW DHT node running a block chain module *thin* node. Both plots show the linear and logarithmic fit to the data. (a) only shows mean values, while (b) shows also all the sampled values. 103

8.9 RAW DHT average start-up times. Full bars represent the time from the moment the software is started to the moment the DHT client is ready to manage user requests (or ready as a back-end). Each bar is broken down in time spent starting the DHT module, the block chain module and initializing other software data structures e.g. database, logging facilities, and other transient Java objects. 105

8.10 Plot showing the number of of active nodes during evolution of a network made up of a maximum number of 8 or 32 nodes, each of them equipped with a *thick* block chain client. The two plots shows different rates of nodes reshuffling. 108
8.11 The red points mark the average store times recorded during the churning networks experiments. Blue points instead marks the average store times (both from Figure 8.4 and Figure 8.5). Note that the dashed vertical lines are drawn exclusively to guide the eye: particularly in the left side of the graph points coupling may be mistaken. 109

List of Tables

8.1 Eridano cluster system specifications. 90
8.2 System specifications of personal computers used to test RAW DHT software. 92
8.3 Duration of the experiments performed running RAW DHT software. 93
8.4 This table shows, for every "churning network" experimental setup, both the average percentage of nodes that failed during the distributed software run and the maximum percentage of nodes failing during any of such experiments. Note that during each of the experiments the network evolved (nodes leaving or re-joining it) randomly. 106

List of Acronyms

CA Central Authority 31

CAN Content Addressable Netwgeork 15

CLI Command Line Interface 78

CS Certification Service 43

DNS Domain Name System 67

DoS Denial of Service 44

DDoS Distributed Denial of Service 39

DHT Distributed Hash Table ix

DOLR Decentralized Object Location and Routing 11

ESB Enterprise Service Bus 75

F2F friend-to-friend 25

HSQLDB HyperSQL DataBase 78

IBS Identity Based Signature 40

IP Internet Protocol 11

ISP Internet Service Provider 83

JVM Java Virtual Machine 77

MANET Mobile Ad-hoc Network 30

MOM Message Oriented Middleware 75

NAT Network Address Translation 91

OLS Ordinary Least Squares 97

ORB Object Request Broker 75

ORM Object-Relational Mapping 78

P2P peer-to-peer 9

PHP PHP: Hypertext Preprocessor 11

PKG Private Key Generator 40

PKI Public Key Infrastructure 69

RAW DHT Random-Age-Work DHT x

RMI Remote Method Invocation 76

RPC Remote Procedure Call 24

TCP Transmission Control Protocol 81

TDD Test-Driven Development 77

UDP User Datagram Protocol 81

VANET Vehicular Ad-hoc Network 30

Colophon

Some of the illustrations created for this document use icons made by Alfredo Hernandez[1] and by FreePik[2] from `www.flaticon.com` and licensed under the Flaticon Basic License.

This document was typeset using LaTeX and the memoir class created by Peter Wilson. The original template was created by Federico Maggi (fede@maggi.cc) with extensive modifications by Vel (vel@latextemplates.com).

[1] `www.flaticon.com/authors/alfredo-hernandez`

[2] `www.flaticon.com/authors/freepik` and `www.freepik.com/`

Introduction 1

We introduce a new *Distributed Hash Table* (DHT) system concept, resilient to attacks against keys integrity. We will go through motivations that lead to this work, a general description of DHTs, and afterwards we will move on to a design that prevents malicious entities from tampering on selected keys. Then we describe a possible implementation of this design, backed up by a specifically repurposed block chain system. Need for a tailored block chain, instead of piggybacking our system on an already existent one, will also be addressed. Before diving in the details presented through this body of work, next paragraphs will provide the reader with a general overview.

Network pervasiveness makes distributed software a modern and attractive opportunity for developers. Cloud computing systems are usually provided by companies that – by means of applications, platforms and infrastructures – enable final users to access ubiquitous services. Also, high bandwidth connection availability and the growing computational capacity of PCs as well as hand-held devices can contribute to the rise of spontaneous and heterogeneous distributed systems. Especially in the open source community, development of such systems could lead to new and unprecedented interesting applications.

Distributed software development however, contrarily to the more traditional centralized software architecture, comes with a few complications due to its network and variable nature. To help

developers cope with data exchange among the spreaded instances issue – i.e. data marshalling – many middleware resources has been released over the years. Two representative examples are CORBA or Java RMI; for a more complete list of such tools please refer to Section 7.1. Letting aside considerations about the required (often steep) learning curve associated with these instruments, the issue of the application network setup is almost always neglected and left to the ingenuity of the final developers. Albeit some aspects of the network setup are straightforward – e.g. socket management – dealing with spontaneous unstructured networks requires, for example, discovery of new idling application instances and routing within the distributed overlay. These issues may prove absolutely not trivial in practice.

Distributed Hash Tables (DHTs) are a popular choice addressing both routing and node location. A general DHT is a distributed system that, ultimately, resembles a normal hash table data structure but allows each of its instances to store key/value records on its network and retrieve them in a second moment. These records are distributed among the instances using some kind of metric between the record's key and nodes *identifiers* (IDs). Resources (whose identifier usually coincide with the key's hash) and node's identifiers share the same address space. Many different DHT schemes have been proposed over the years: the more important designs are described in Section 2.3. To overcome small unimportant differences, we outlined a general DHT model, presented in Section 2.2. In this work DHT records will always be described as key/value couples even if, for example, Chord (Section 2.3.2) only manages keys while KAD (a Kademlia implementation, Section 2.3.5.1) links keys with references of nodes owning resources identified by certain keys. A key/value abstraction is the most general possible: in Chord's case value can be considered a "null" value and in KAD's case it is just a proxy to the actual value. Due to their loosely knit but highly organized overlay, DHTs allow every node to route over a changing network requiring a small number of contacts compared to the overall network size. Each node is required to host a relatively small set of keys but nonetheless record integrity is generally granted. Among all DHT designs, those based on Kademlia are nowadays the most used, backboning popular file-sharing systems like eMule and Bittorrent (plus their variants). These systems serves millions of users, and routing is almost entirely based on their DHTs.

DHTs, unfortunately, are known to be prone to a number of security threats. A key aspect of DHTs is versatility, a trait provided by its self-organizing structure. However such versatility is favored by lack of nodes authentication: almost all current DHT designs completely trust nodes' information provided by nodes themselves. This feature can be exploited to perform various attacks, the most famous being the *Sybil attack* (Section 3.1) – which was unveiled just a year after the first DHT designs was presented. *Sybil attack* however is not the only threat, nor necessarily the most important one. In Section 3.2 and Section 3.3, for example, we present two attacks that directly undermine records availability: *Node insertion attack* and *Publish attack*. In a *Node insertion attack* a malicious node purposely chooses its ID; this way it persuades the other nodes to bestow on it storage of a specific key. Then, whenever an honest DHT instance will ask for the record identified by target key, malicious node will simply deny it or, alternatively, reply with bogus information. Performing such attacks a malicious entity can easily target specific nodes, keys, values, or all of the aforementioned resources. Chapter 3 extensively presents the major security threats impairing Distributed Hash Tables.

Nonetheless a reliable *and* secure content-based or resource-based routing layer appears to be essential. Widely deployed applications – e.g. Bittorrent – still rely on DHTs in spite of their vulnerabilities due to the fact that no valid alternative to DHTs tackling the issue of dynamic resource routing is known. Reliable countermeasures for these attacks are still considered open questions. Security weaknesses limit DHTs to be just "best effort" systems and make them only suitable to some kinds of applications. However even more security-aware applications could (and would) benefit from what a DHT has to offer: aside from a network layer where nodes may be located relying on dynamic keys rather than physical network addresses, also the possibility to store short chunk of data on an ubiquitous medium could help developing applications coping with fast changing deployment scenarios.

An intuition on how to counteract *Node insertion attack* however may reside in a non-fixed resource positioning. To successfully gain control over a certain resource, the attacker must carefully choose its "position" – i.e. its ID – trying to become the "closest node possible" to its target. If keys were to periodically "move" (or "rotate") across the resources' address space along unpredictable trajectories though,

such an endeavor would get dramatically harder. So, creating a mechanism that permits to permanently keep resources "on step ahead" of malicious nodes could be a way to frustrate any attacker's attempt to tamper with a target resource.

Elaborating on this intuition we define a new DHT that, relying on the addition of just two new primitives, leads to a system protecting key integrity as well as limiting the appearance of *Sybil* nodes. This new design, which is presented in Chapter 4 is built upon the generic DHT system of Section 2.2 retaining all of its basic primitives. On top of them we define *getSeed* and *isOldWorker*. To protect keys integrity *getSeed* and *isOldWorker* do work together. The first, *getSeed*, provides access to a shared random seed for every node in the network. This seed will, of course, periodically change and all the network accessing *getSeed* will be able to access its current value – the correct one. Nodes will then combine such seed with the normal mechanism distributing keys among the nodes partaking to the DHT. Changes in the random seed will affect resources positioning among the DHT nodes, thus causing a periodic "resources' rotation". The other primitive, *isOldWorker*, checks if an ID become active *before* current random seed was generated and that, in the recent past, it invested some effort in the task of maintaining incepted ID. Thus whenever trying to store a resource on a node (through its ID), using the combination of *getSeed* and *isOld*, recipient will be considered trustworthy if and only if:

1. recipient node's ID appears to be the closest to the (randomly generated) resource's position

2. recipient node chose its ID before anyone could know that resources position would be close to such ID

3. recipient node spent some effort to "activate" its ID.

Requiring an effort at ID inception time is necessary to discourage nodes from, slowly but steadily, generating as much IDs as necessary to eventually rise their probability of "reaching the correct location". Moreover the number of different IDs behind which a single node can "hide" is bounded by the ability of the offending node to create them. A simple way to achieve this lies in requiring each node to provide a difficult to compute yet easy to check token derived from *getSeed* current value. The set of new primitives can be

incorporated in a generic DHT design with minor modifications on the basic primitives mechanisms – e.g. resources hashing and routing procedures. Chapter 5 provides proofs that this newly proposed proposed design is correct and secure.

The main challenge in our new DHT design resides in generating and accessing a distributed random seed. The two new primitives proposed by this work rely on providing all the network nodes with a consistent value. The problem of distributed consensus however is still an open one [29]. Nonetheless if, instead of a global consensus, we loose requirements to a "quasi-consensus" or to a "almost-distributed-consensus" scheme, tractability of the issue significantly improves. Example of systems relying on almost distributed consensus are block chain systems.

Block chains (a general description is available in Section 6.1) are a distributed data collection, whose security is based on the difficulty of chaining new data to existing one. Data is organized in quantum units called *blocks*, linked one another building up a chain. A *block* is identified both by its position in the chain and by a value obtained hashing it. The difficulty securing the system lies in the unpredictability of required data fragments without which new *blocks* are not considered valid: to the very least a *block* must satisfy certain criterion on its own hash *and* include the hash of its predecessor *block* among the data contributing to its own hash value. Block chains are known to undergo transient disagreement about the newly added chunks. Nonetheless, disregarding the most recent *blocks* added to the chain, eventually all participating nodes converge to a network-wise consensus about the main trunk stored in the system.

To back up our new theoretical DHT design we developed a working proof-of-concept implementation. This system is coded in Java and has a modular architecture. The software separates the local system management operations; a database interface; a module providing a back-end for the two new primitives; and a module managing DHT operations as well as an interface to access it. Our implementation is described in detail in Chapter 7 while Chapter 8 discuss its practical performance analyzing the system through experimental evaluation.

The module serving as back-end for *getSeed* and *isOldWorker* implements a customized block chain. Unlike popular existent block chains – e.g. BitCoin, detailed in Section 6.3.1 – ours does not have a

coin value associated: the *transactions* stored in its *blocks* do record authentication credentials linked to DHT identifiers (IDs). Aside from this difference and the choice of SHA-512 hashing algorithm, our block chain module is modeled after BitCoin under every other conceptual aspect.

The DHT module is an implementation of the design proposed in Chapter 4. It is a Kademlia-like system (see Section 2.3.5 for details about Kademlia) using the block chain module as source of random seeds and as "registration authority". In a nutshell both *getSeed* and *isOldWorker* can be easily piggybacked on a block chain due to its very nature. A *block*'s hash can be used as a random seed: there is no possible way to know beforehand the value of the next *block*'s hash, independently from of how much information about the already known *blocks* is available. To tune the new seeds extraction rate one can tune the difficulty to computer a new *block* or (preferably) devise a mechanism selecting only some of the chained blocks; for example a seed could be represented by the value of the hash of a *block* whose number is divisible by a certain prime number. Once established which *blocks* – namely the "seed *blocks*"– will yield a new seed, *getSeed* can be implemented by just accessing the current block chain copy and returning the hash of the most recent seed *block* available. Thus said implementing *isOldWorker* is even simpler: an ID will pass *isOldWorker* check if it appears in a *transaction* stored in a *block* preceding the one used by *getSeed*. We also require that any node provides in its *transaction* an integer that hashed with its own ID and the value of the submission's seed obtained from *getSeed* produces an output satisfying some constrains: finding such an integer may require an extensive sequential search, but its verification is straightforward. Additionally a constrain about "how old a *transaction* can be" is required, but this will be detailed in Chapters 4 and 5.

Usage of a block chain is a convenient solution, both for its out-of-the-box nature and for its replaceability. Cryptocurrencies stems from very thriving communities, made up by academics as well as practitioners. For their economic value (see Section 6.3.2) bugs, vulnerabilities and weaknesses are constantly discovered and tackled down. Thus we can concentrate on the problem at hand – make a secure DHT – trusting that our back-end is not a criticality by itself or that, whenever a problem may be encountered, the community will readily hand over a quick fix. Moreover it is important to note

that a block chain is *just a possible* choice to implement a back-end for our secure DHT: it is not, by any possible mean, the only solution nor necessarily the best one. Any (distributed) mechanism providing

1. a random value accepted by the majority of the network
2. a way to timestamp an ID inception and its required effort

will be equally acceptable.

We chose to develop a new block chain instead of using an existent one out of necessity. There exist block chains that, alongside information about coins ownership, store custom data retrievable through specialized clients. One of the most prominent example of such systems is NameCoin (more details on it can be found in Section 6.3.3). Albeit NameCoin offers the possibility to store personal records, a plain text representation of the authentication data required by our system exceeds the maximum size allowed by this block chain. Moreover players entering the speculation block chain business may cause unsolicited surges in transactions that, although not affecting the system functionality, could lead to a futile momentary slow down.

Concluding, our contribution with this work is threefold, theoretical and practical. We describe two new primitives and proof their validity; on these new primitives we describe a novel actionable DHT design, generic enough to be either implemented on its own or to be applied to already existent designs with minor adjustments; and finally we developed a modified version of a Kademlia DHT that, using a block chain, implements our new secure design to empirically validate our proposed system.

The Distributed Hash Table 2

In this chapter the *Distributed Hash Table* (DHT) concept will be introduced. In the first section we provide a general description and motivation for DHTs while in the second section we will present a general DHT model. This model will be extended in Chapter 4, where we outline a secure Distributed Hash Table design. The last section of this chapter will describe in detail some notable DHTs, giving the reader a better understanding of the most popular designs in this field.

2.1 Description

The first part of this section provides a brief history that lead to the development of DHTs and their motivation, while the second part gives a general idea of what a DHT is and behaves.

Large heterogeneous (possibly spontaneous) distributed systems, like for example *peer-to-peer* (P2P) systems, do have the critical need of locating any kind of resource required by the system. These include, for example, system nodes, data records, file pointers or any other information disseminated across the system itself.

By the end of the 90s were *Napster*, *Gnutella* and *Freenet*[1] were popular P2P systems. Although they all shared similar goals these

[1] *Freenet*, unlike original *Gnutella* and *Napster* clients, is still developed and available at `http://freenetproject.org/`

systems used different protocols to locate data, hosted on the peer nodes composing the network. Following paragraphs provide an overview both of these systems and how they dealt with the issue.

Napster[2] was (mainly) a music file-sharing platform and, probably, the first large P2P content delivery system. Resources' indexing was performed by a central server on behalf of the peers. When joining the network each node would sent the server a list of the files it hosted. Thus when nodes searched for a resource, this query was managed by the central server that would then reply to it with a list of nodes advertising ownership of requested files. The querying node could then contact one or more of the discovered nodes and directly download the file of interest. The central server feature represented a single point of failure which exposed *Napster* to hostile attacks or lawsuits and, ultimately, deemed its fate [13].

The name *Gnutella* may refer both to *Gnutella protocol* and to the first *Gnutella client*, from whom the network derives its name. The first client, originally developed by Nullsoft was retired soon after its initial release, but in a matter of few days *Gnutella protocol* was reverse engineered and a number of free open source compatible clients made their appearance. *Gnutella protocol*, unlike *Napster*'s, did locate resources on the network through a flood query strategy. Albeit flooding the network in an attempt to locate a resource does avoid a single point of failure, it is obviously less efficient. Since *Gnutella protocol* does rely on a totally unstructured network overlay, it cannot be easily shut down like happened with *Napster*. For this reason a number of clients are currently developed and maintained.

Freenet [17] is a fully decentralized free software allowing users to share whole files. These files are stored in a distributed file-system fashion and are associated to keys. The system lets users retrieve desired files through a key-based routing heuristic. However no guarantee that data will be found and retrieved is given. As a side note, *Freenet* allows the creation of web sites (called "freesites") hosted and reachable through its network. Note that "freesites" can contain only static content and, due to bandwidth and latency lim-

[2]The original *Napster* system ceased operations in 2001 due to legal difficulties over copyright infringement. It was then acquired by *Roxio* that, using the same name and logo, turned it in an online music store. In 2011 the *Napster* store was merged with *Rhapsody*.

itations, complex (albeit extremely common) web technologies like *PHP: Hypertext Preprocessor* (PHP) or *MySQL* are not supported.

Even if *Gnutella*'s and *Napster*'s goals differed from *Freenet*'s, the three systems all shared a common necessity: to publish (and, of course, to locate and retrieve) data on their overlay networks. This is a non-trivial task: system's nodes can usually be reached only via their *Internet Protocol* (IP) address.

Need for an infrastructure that can provide a resources look-up service or a *Decentralized Object Location and Routing* (DOLR) is what motivated development the first DHTs. A DHT offers such a service letting the nodes store and search (*key*, *value*) pairs as in an *hash table* data structure, but these records are distributed (and possibly redounded) throughout all the nodes participating in the system's network.

Distributed Hash Tables use a structured key-based routing mechanism attaining the decentralization feature of *Gnutella* and *Freenet* but still guaranteeing accurate and efficient results as *Napster* did.

Usually DHTs are characterized by three main properties: the system is built up from the nodes without central coordination – i.e. autonomous and decentralized; system reliability, within reasonable constrains, should not be affected by nodes joining and leaving the network or simply failing – i.e. fault tolerant; system efficiency should be independent by network's growth in the number of nodes – i.e. scalable. A common technique used by most DHTs to achieve these features is that each node is required to coordinate with a small fraction of the nodes in the network; usually each node know a small fraction of the n total nodes, typically just $O(\log n)$.

In the next section we provide a model for a basic DHT. In Section 2.3 instead some notable DHTs will be described.

2.2 A basic DHT model

In this section we will describe the primitives and objects required for a generic DHT. Through the years the issue of dynamically and efficiently locate a particular resource – e.g. a key, a reference to a file or a reference to a node – has been addressed in multiple ways, most notably via Distributed Hash Tables. Despite differences from one design to another, a common desired behavior emerges

describing what could be considered an abstraction for a "basic distributed hash table". Such a DHT would provide a system where participants can store data records relying on other nodes and retrieve them.

Though Kademlia [63] and Chord [85] are probably the two best known DHTs also CAN [76], Pastry [79] and Tapestry [102] are among the first and seminal works that opened this research field. All of these design share some basic fundamental operations such as *looking up* for a key (or node) identifier (*ID*), *storing* a key/value pair on the network, and a *bootstrap* procedure. Moreover all nodes participating to the DHT will need to upkeep a *routing table* (necessary to find other nodes and resources), must have a global agreement on how data keys will be processed by means of an *hash* function, and a *distance function* is defined for *IDs*. As a final note, beside the primitive operations already described, the network will need a procedure for nodes to join and leave the system; also some kind of deletion policy is required for all stored data.

In the next paragraphs a more detailed description of each component sketched out so far will be provided.

On the DHT each unit of data, or *resource*, is in the form of a key/value record. Please note that this is abstraction is the most general definition possible as it can describe a number of different real-life scenarios: while key is a fundamental feature, value may be absent (or "null"), a simple reference (e.g. a proxy to some actual resource), or an actual value indexed by its associated key. Identifiers (*IDs*) are used to address *resources* and nodes alike and share the same address space. All DHT nodes are required to use an *hash* function converting keys to their *IDs* and, at the same time, uniformly distribute the resources' *IDs* on the address space. All possible *IDs* are totally ordered through a *distance function* defined on the address space itself.

Every node maintain a *routing table* listing a set of other nodes organized according to the *distance function*. The network view created by the *routing table* allows a node to identify the closest node (according to the *distance function*) to a desired *ID*. The *routing table* fundamental feature to search resources. To upkeep a *routing table* continuous update are required, finding more suitable nodes and/or purging the not active ones. To achieve an efficient table maintenance different strategies can be put in place. Without loss of generality we model it as a *ping* protocol, executed by each

node thus refreshing the local *routing table*.

Bootstrap is a primitive that provides a starting node with a set containing at least one DHT active node. To join the network a new node must at least contact (therefore know) at least one other participant to the network. The node *bootstrapping* can thus start building up its network view populating its *routing table* with fresh contacts.

Look-up is a crucial function to a DHT system. It locates (through the local node *routing table* and, if necessary, by querying other known nodes) the network address of the node whose *ID* is the closest, accordingly to the *distance function*, to a searched target *ID*. A *look-up* can be issued to locate a certain node or resource (both to store a record tied to such a resource or to retrieve it).

Store is used by a node to ask the DHT network to save a key/value record. A node wishing to *store* a certain record, performs a *look-up* with the record's key thus finding the node (or set of nodes) to whom the *store* request should to be submitted. Each DHT node hosts a local *hash table*, a data structure that indexes values according to their keys hash values, where records received via *store* queries are kept. A *store* request is expected to receive a boolean reply – confirming or denying the request outcome. If no reply is received, the node that issued the *store* request will assume that its query failed. Although not all the major DHT designs or implementations [63, 84, 85, 76, 79, 102, 56] share the same approach on stored records removal, there appear to be a consensus on the fact that a record is expected to eventually disappear from the DHT network. A convenient mechanic to achieve this feature is to impose a time threshold: after a period of time a resource is either renewed through a new *store* request or it simply expires.

A *find value* primitive works (similarly to *store*) performing a preliminary *look-up* for a given resource's key *ID*, locating the node (or set of nodes) where the searched resource is expected to be stored and, if possible, retrieving the resource's value. If the located nodes do not have in their *hash table* any record matching searched key, a special value – e.g. "null" – is returned to the request sender.

Nodes can *join* and *leave* the DHT network at any time. It is responsibility of a newcomer node to report in when it come online for the first time or starts interacting with a node never contacted before. Thus other nodes are enabled to add it to their own *routing tables*. Leave *join* responsibility to new nodes eases the necessity for

a cumbersome distributed protocol. On the other hand the *routing table*'s eviction policy of inactive nodes makes superfluous a *leave* protocol (still, is a non-trivial issue).

Usually DHT architectures bound the number of nodes stored in the *routing table* – or contacted during *look-up*, *find value*, and *store* communications – to a factor logarithmic in the number of network's the total nodes. Although this is an interesting and elegant feature, we will not focus on it since it does not really relate with the DHT vulnerabilities examined in this paper.

Summarizing, a general DHT should have the following primitives: a *bootstrap* providing a new node with the information needed to *join* the network; a *look-up* that from a certain *ID* allows to to locate some nodes; a *find value* to retrieve, if any, the value(s) associated to a certain resource's key; and a *store* procedure to save on the network a resource's key/value record. Helping these primitives two data structures (the nodes' *routing table* and *hash table*) and two functions (a *distance function* and an *hash function*) are required. Using these basic components we can model a basic Distributed Hash Table. It is easy to incorporate in this design a replication policy, increasing resources availability: whenever *look-up*, *find value* are invoked up to k nodes are returned and *store* queries will be performed to k nodes as well.

However, as will be explained in Chapter 3, a variety of attacks – e.g.: *Node insertion* or *Sybil attacks*, see respectively Section 3.2 and Section 3.1 – can be performed with no considerable effort in such a basic DHT. Through these attacks an adversary can easily perform a selective censorship or deny the resources stored on the DHT.

As a final note, throughout this entire work we assume that any network communication has a timeout, thus avoiding problems deriving from a node incapacitated to reply a query due both to node overload or to node failure.

2.3 DHT examples

This section will present the most famous DHTs that ignited the interest in this particular research field. For each DHT the key features will be described giving the reader a flavor of the various design possibilities. Moreover comparing each of these designs with

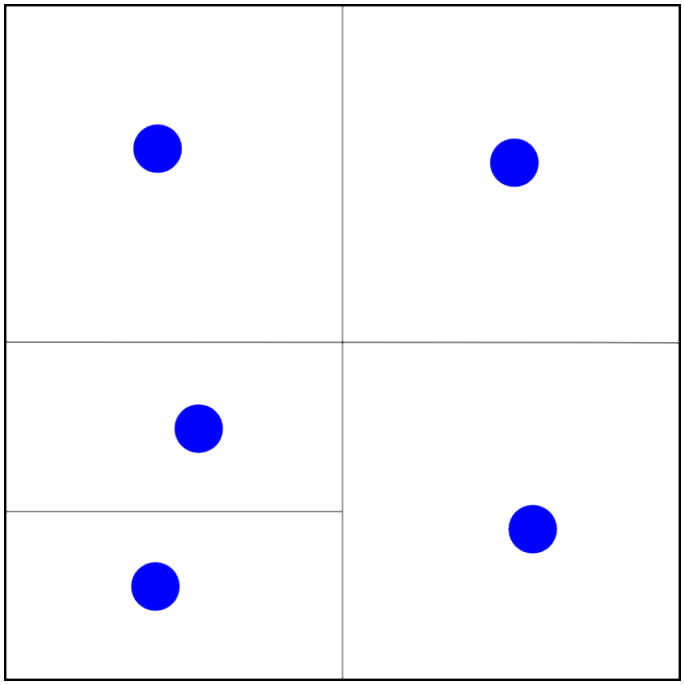


FIGURE 2.1: Example of a CAN 2-d virtual space partitioned among five nodes.

one another helps to understand how the general DHT described in previous Section 2.2 can model actual implementations.

2.3.1 CAN

Content Addressable Netwgeork (CAN) [76] was presented as a distributed, Internet-scale, hash table. Its design centers around a virtual d-dimensional Cartesian coordinate space. CAN's coordinate space is purely logical and bears no relation to any physical coordinate system. The entire coordinate space is dynamically partitioned among all the nodes in the system such that every node is responsible for its individual, distinct zone within the overall space. An example of this partitioning is shown in Figure 2.1.

Storage of (*key*, *value*) pair in the virtual coordinate space happens as follows: (K_1, V_1) pair's key K_1 is deterministically mapped onto a point P in the coordinate space using a uniform hash function. Once located the node responsible for the zone within which point P lies, (K_1, V_1) is sent to it for storage. To

retrieve an entry corresponding to key K_1, any node can apply the same hash function to map K_1 identifying point P. Thus a node can then retrieve the corresponding value routing the request through the CAN infrastructure until it reaches the node in whose zone P lies.

In CAN nodes keep a local routing table that holds IP addresses linked to coordinates (on the virtual space) of its immediate neighbors. In a d-dimensional coordinate space two nodes are considered neighbors if their coordinate spans overlap along $d - 1$ dimensions and is adjoining along the remaining dimension.

A CAN node moreover holds information about the "adjacent" zones of the global hash table to allow fault tolerance. This way whenever a node leaves or fails another CAN node, who already stores the required information, is able to take charge of the zone left uncovered.

The set of primitives described in CAN's original paper [76] are:

- a *bootstrap* allowing a new node to discover IP addresses of other nodes currently in the system
- a *join* procedure allowing a node to enter the network and take charge of a zone of the virtual space – whenever a new node logs in the CAN network, its new neighbors release control over the virtual space's area claimed by it
- a *leave* procedure used by a node during a graceful shutdown to hand over its zone and associated records to one of its neighbors
- a *takeover* procedure used after discovering a node's failure and allowing one of its neighbors to take charge of its zone
- *look-up* of a key, allowing a node to find the node responsible for a (*key*, *value*) record
- *insertion* of a (*key*, *value*) record, storing it onto the correct CAN node
- *deletion* of a (*key*, *value*) record, removing it from the global hash table maintained by CAN.

2.3.2 Chord

Chord [85] was presented in 2001 as a protocol supporting just one operation: given a key, map such key onto a node. More sophisticated operations like data location or (*key*, *value*) records storage were not discussed in the original paper and left as a conceptually easy extension on top of Chord itself.

Chord uses *consistent hashing* to attain (with high probability) load balancing: all nodes will be responsible, roughly, of the same number of stored keys. The consistent hash function assigns each node and key an m-bit *identifier* using a common hash function such as SHA-1 [23]. *Identifiers* are ordered and organized in an *identifier circle* modulo 2^m. Key k is assigned to the first node whose identifier is equal to or follows k's identifier in the circle representing the identifiers' space. This node is called the "successor node" of key k and denoted as $successor(k)$. If identifiers are represented as a circle of numbers from 0 to $2^m - 1$, then $successor(k)$ is the first node whose identifier can be find traversing the circle clockwise starting from k.

The *successor* concept applies to nodes as well, and is complemented by the *predecessor* concept. *Successor* of a node is, like for keys, the first node encountered in the *identifier circle* in clockwise direction. The *predecessor* instead is the first node encountered on the circle going counter-clockwise.

Each Chord node keeps a *finger table* containing up to m entries. The i^{th} entry of node n's *finger table* store the information needed to reach over the network $successor((n + 2^{i-1}) \mod 2^m)$. In addition to the *finger table* a node also maintains a *predecessor pointer* containing IP and *identifier* of the immediate predecessor. An example of these pointers (*fingers* and *predecessor pointer*) is depicted in Figure 2.2.

To *look-up* a key k a node will search its *finger table* for the closest $successor(k)$ known. The initiator node then will pass the query to found successor who will reiterate the procedure until a node finds out the key is stored by its immediate successor. Using this mechanism based on *finger tables*, the expected number of nodes that must be contacted to locate a key in an N-node network is $O(\log N)$.

When a new node joins the system, within the Chord network the following is expected to happen: the new node must initialize

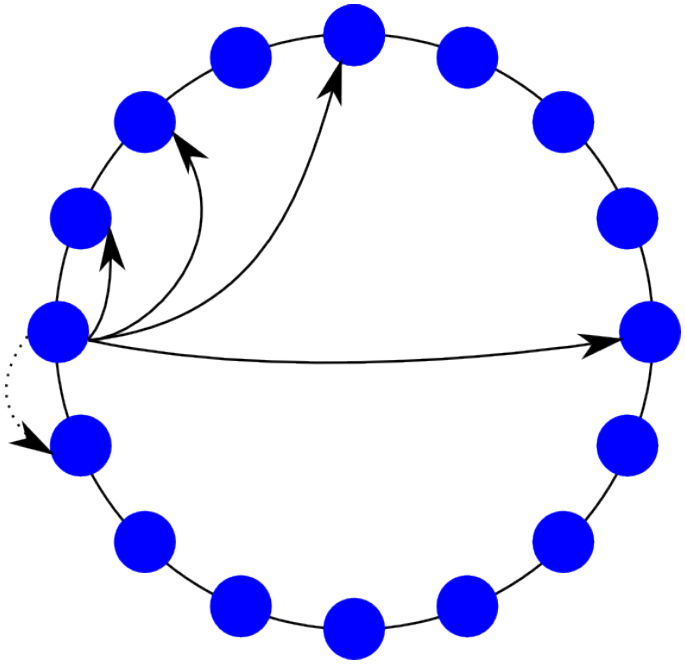


Figure 2.2: Example of a 16 nodes Chord network. One of the nodes shows its *fingers* and its *predecessor pointer*. Solid arrows (inside the *identifier circle*) represent *fingers* to the other nodes, while the dotted arrow (outside of the *identifier circle*) represents the *predecessor pointer*.

its *predecessor pointer* and *finger table*; existent nodes must update their *fingers* and *predecessors* accordingly; new node's predecessor transfer to the new node all the keys that now are closer to it; optionally nodes can notify a higher layer software relying on Chord routing.

Chord's original paper [85] describes the following primitives:

- a *join* procedure, allowing new nodes to join the network and letting existent nodes to update they *finger tables* and *predecessor pointers*
- *look-up* of a key k locating the node being the k's *identifier* immediate successor in the *identifier circle*
- a *stabilization* protocol ensuring *fingers* and *predecessors* pointers of every nodes are kept up-to-date, allowing fault

tolerance in the case of nodes failing or leaving the network.

2.3.3 Pastry

A Pastry [79] system is a self-organizing overlay network of nodes, where each node routes client requests interacting with local instances of one or more higher layer applications. Each node gets a 128-bit integer used as node *identifier* (*nodeId*) and marking the node's position in a circular *nodeId* space ranging from 0 to $2^{128} - 1$.

The *nodeId* is assigned randomly when a node joins the system. In Pastry's original work is assumed that *nodeIds* are generated such that the resulting set is uniformly distributed across the *nodeId* space [79]. This way, with high probability, nodes that are close in the address space – e.g. have adjacent *nodeIds* – reside in machines that are geographically separated.

For routing purposes, *nodeIds* and key's identifiers are thought of as a sequence of digits with base 2^b (where b is a configuration parameter, typically with value 4). Please note that in the following we use the term "key" and "key's identifier" interchangeably. Pastry routes messages to the node whose *nodeId* is numerically closest to the given key's identifier, forwarding messages from one node to another. A node forwards a message to a node whose *nodeId* shares with the key a prefix at least one digit (b bits) longer than the prefix shared by the key and the forwarding node's *nodeId*. If no such node is known, the message is forwarded to a node whose *nodeId* shares a prefix with the key as long as the current node, but is numerically closer to the key than the present node's ID. If no other possible forwarding is possible, routing is complete.

The routing overlay network is built on top of the hash table by each peer discovering new nodes and exchanging state information consisting of a list of *leaf nodes*, a *neighborhood list*, and a *routing table*. It is important to note that in a Pastry network it is assumed the existence of a *proximity metric* which should be provided to Pastry by an higher level application layer.

In a network consisting of N nodes, a *routing table* is organized into $\lceil \log_{2^b} N \rceil$ rows, containing up to $2^b - 1$ entries. Each of such entries is a reference to a node whose *nodeId* shares the current node's *nodeId* the first n digits, but whose $(n+1)$-th digit has one of the $2^b - 1$ other possible values rather than the $(n+1)$-th digit of the current node's *nodeId*.

The *neighborhood list* M is a set containing *nodeIds* and IP addresses of the $|M|$ nodes that are closest (according to the *proximity metric*) to the local node. Although it is not used directly in the routing algorithm, the neighborhood list is used for maintaining locality principles in the routing table. Typical values for $|M|$ are 2^b or $2 \cdot 2^b$.

The *leaf nodes* list L consists of the $|L|/2$ numerically closest peers whose *nodeId* is larger than the local node's, and of the $|L|/2$ nodes with smaller *nodeId*. Typical values for $|L|$ are 2^b or $2 \cdot 2^b$ as well.

In the event of a failed node in the *leaf nodes* list a node contacts the live node with the largest index on the side of the failed node, and asks that node for its *leaf table*. This procedure guarantees that each node lazily repairs its *leaf set* unless $\lfloor |L|/2 \rfloor$ nodes with adjacent *nodeIds* have failed simultaneously.

The primitives that Pastry is supposed to expose as described in [79] are just the following two:

- *pastryInit* that causes the local node to *join* the network and initialize all its internal structures
- *route(msg, key)* causing Pastry to route *msg* message to the node whose *nodeId* is numerically closest to the *key*.

On top of those, applications layering on Pastry should manage:

- *deliver(msg, key)* called by Pastry to notify message *msg* to a node whose *nodeId* is the closest to *key*
- *forward(msg, key, nextId)* notifying local node that message *msg* addressed to the closest node to *key* is about to be forwarded to a node identified by *nextId*. Local node can change the content of the message or terminate its routing at the current node
- *newLeafs(leafSet)* notifying the higher application layer that the local Pastry node's *leaf nodes* list was updated in *leafSet*.

2.3.4 Tapestry

Tapestry [103, 102] is a P2P system offering efficient, scalable, self-repairing, location-aware routing to nearby resources. It is an

extensible infrastructure providing decentralized object location and routing, focusing on efficiency and on minimizing message latency. To achieve this Tapestry constructs locally optimal routing tables from initialization and maintains them trying to reduce routing stretch. Furthermore, Tapestry allows object distribution determination according to the needs of a given application.

Tapestry dynamically maps each identifier G to a unique node, called the identifier's root G_R. If a node N exists with $N_id = G$ then this node is the root of G. At each hop a message is progressively routed closer to G by incremental suffix routing. Each node stores a neighbor map that has multiple levels. Each level contains links to nodes whose ID match with the host node up to a certain digit position. The i^{th} entry in j^{th} level records the ID and location of the closest node whose ID begins with $prefix(N, j-1) + i$: this means that level 1 contains entries that have nothing in common in their IDs, level 2 contains references to nodes whose ID has the first digit in common, and so on.

To publish an object participants in the network periodically route a *publish message* toward the root node for that object ID. Each node along the path stores a pointer mapping the object. An object (or node) is then located by routing a message towards the root of the object's (or node's) ID along the path, checking the mapping and redirecting the request appropriately.

A new node joining the network becomes the root for its own *nodeID*. After its inception, the existing root finds the length of the longest prefix of the ID it shares and sends a multicast message that reaches all existing nodes sharing the same prefix. These nodes then add the new node to their routing tables and the new node may take over becoming the root for some of the objects that was stored by the old root.

To leave the network, a node broadcasts its intention of leaving and appoints a replacement node for each level in the other nodes' routing tables. Objects held by the leaving node are redistributed or replenished from redundant copies. Unexpected node failure is instead handled through redundancy in the network and backup pointers reestablishing damaged links.

Tapestry [102] describes the following primitives:

- a *join* procedure (part of the *node membership* Tapestry component) allowing a node to enter the network

- a *leave* procedure for nodes to gracefully leave the network
- *routeToObject* and *routeToNode* to find either the root node for a given key or an exact *nodeID* match
- *publishObject* to make available an object at the local node
- *unpublishObject* attempting to remove location mappings of a certain object.

2.3.5 Kademlia

Kademlia [63] is a P2P system routing queries and locating nodes using a XOR-based metric topology. Each Kademlia node has a 160-bit node ID, supposedly chosen as a random identifier when a node joins the network. Node ID is included as part of every message exchanged in the network, thus allowing the recipient to record the sender's existence if necessary. Keys are opaque, 160-bit quantities (e.g., the SHA-1 [23] hash of some larger data).

To publish and find *(key, value)* pairs, Kademlia relies on a notion of distance between two identifiers: given two 160-bit identifiers, x and y the distance between them is defined as their bit-wise exclusive or (XOR) interpreted as an integer, $d(x, y) = x \oplus y$.

To route messages each node stores a routing table holding for each $0 \leq i < 160$ a list of records containing the physical address and node ID for nodes having distance between 2^i and 2^{i+1} from itself. These lists are called k-buckets containing at most k entries, where k is a system-wide replication parameter.

When *looking up* for an ID, searching node picks α nodes (being α a system-wide parameter) from the appropriate k-bucket (if such bucket has fewer than α entries, the α closest to ID in XOR metric are chosen among all the known nodes). To each of these nodes a *find node* request is sent for the searched ID. Each of these nodes (if still active) will reply with a set of α nodes closest to the searched ID. Recursively the initiator re-sends a *find node* request until no new node contacts are found.

To store and find values associated to a given key a *store* or *find value* is sent to the (up to) k closest nodes found via *find node look-up*.

Assuming that every k-bucket contains at least one node, a *look-up* procedure is correct in logarithmic time. A *look-up* procedure

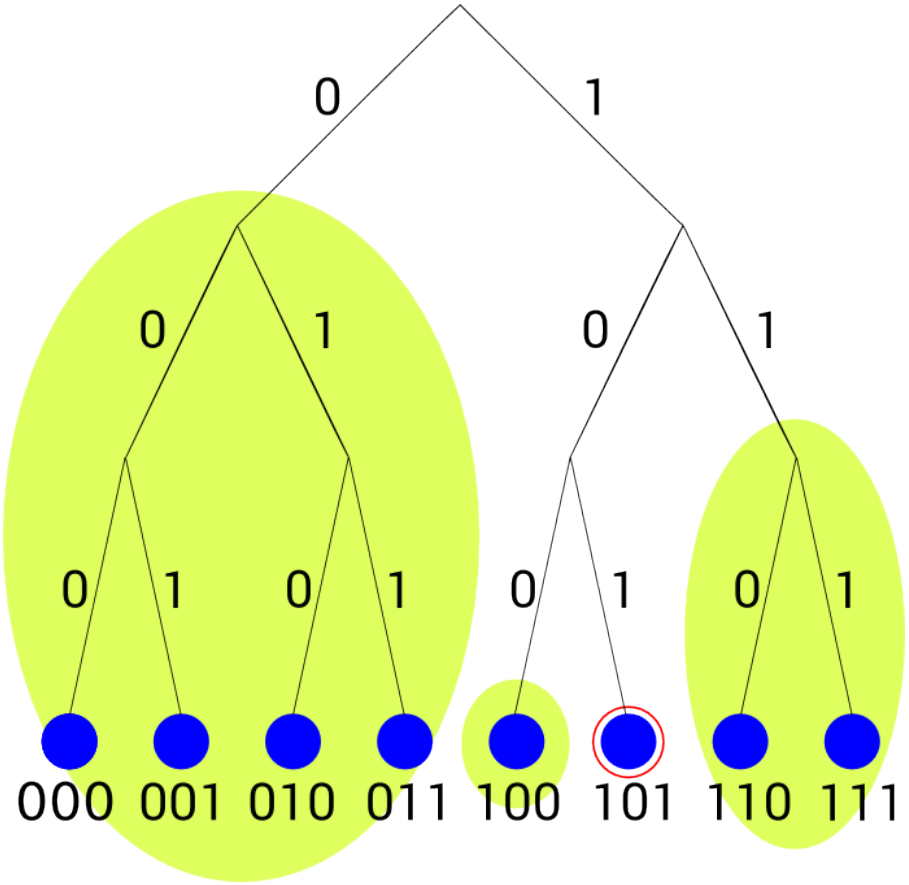


FIGURE 2.3: Example of a Kademlia network using 3-bits node IDs. All nodes from each of the three highlighted areas will be referenced in the same k-bucket in the selected node's (with ID 101) routing table.

will (in the worst case) find at each step a node half as close to the searched ID. Figure 2.3 can help visualizing how distance halves at each step.

A lest-recently seen eviction policy is implemented in k-buckets: when a node receives any message from another node it updates the appropriate k-bucket for the sender's node ID moving it at the top of the bucket or adding it to the bucket if it is not already present. If the k-bucket is full when a new node is added to it, the least-recently seen node – i.e. the one at the bottom of the list – is removed.

In Kademlia, instead of enabling any node to selectively delete (*key*, *value*) records, an expiration time is enforced on any key.

To *join* the network a new node must know the contact to at least one node already participating to the Distributed Hash Table's network. The new-coming node then performs a node *look-up* on its own node ID. By doing so it both populates its own k-buckets and causes the insertion of a reference to itself into the other nodes' appropriate k-buckets.

In the Maymounkov and David Mazières original paper [63] a set of four *Remote Procedure Calls* (RPCs) is described:

- *ping* that probes a node to see if it is online
- *find node* that, given a 160-bit ID argument, finds the k nodes closest to such an ID in XOR metric
- *store* instructing a node to store a (*key*, *value*) record
- *find value* that like *find node* discovers the k closest node to a given ID but, if such k nodes has received a previous *store* instruction for the searched key, returns the stored value.

Moreover, even if not presented as RPCs, the following two primitives are mentioned:

- a *bootstrap* procedure to find at least one node already participating to the Kademlia network
- a *join* procedure to allow a new node both to initialize its own structures and to instruct other nodes about its existence.

2.3.5.1 Kademlia implementations

Unlike other notable DHT proposals, Kademlia has been implemented in a number of major networks, among which the two best known are probably Mainline and the KAD network.

Mainline [56] is the Distributed Hash Table used by BitTorrent[3] clients to find peers, without necessarily use centralized a tracker server. KAD network instead was originally developed by the

[3] `http://www.bittorrent.com/`

eMule[4] community to overcome the usage of eDonkey 2000 server-based network. Both BitTorrent (with it numerous clients) and eMule are well-known file-sharing *peer-to-peer* (P2P) networks, daily accessed by millions of clients. Given the sizes of these two networks, through the years the academic community shown interest in them, performing various experimental measurements both on KAD [84] and Mainline DHT [93, 94].

Other relevant Kademlia implementations have been developed to support a number of different systems. The following are an (incomplete) list of such services: Osiris[5], a freeware program used to create and distribute in a P2P fashion web portals; RetroShare[6], a *friend-to-friend* (F2F) decentralized communication network; InterPlanetary File System (IPFS)[7], a content-addressable P2P distributed file system.

[4] `http://www.emule-project.net/`
[5] `http://www.osiris-sps.org/`
[6] `https://retroshare.github.io/`
[7] `https://ipfs.io/`

DHT Security 3

This chapter presents some of the most important security threats that affect systems that can be described by our basic model of *Distributed Hash Tables* (DHTs) described in Section 2.2; of course the major DHT designs described in Section 2.3 are affected as well.

Next sections will present various attacks exploiting weaknesses derived by the distributed and unstructured nature of DHTs. Each section describes of one of the attacks, its practical relevance and eventually presents the state of the art proposed solutions as well as issues that at the moment are still opened.

Considering that DHTs can be a middle layer for decentralized P2Ps network system, vulnerability to the attacks described in this chapter may limit their efficiency an could even pose a serious threat to freedom of information or speech. On top of that, reducing DHTs's ability to provide an acceptable quality of service, these attacks may also undermine effectiveness of those systems built upon Distributed Hash Tables, thus potentially leading to their abandonment. Nonetheless, DHTs could be an interesting way to provide other applications with out-of-the-box functions like (but non limited to) an easy way to advertise existence of available idling instances and, of course, a routing based on dynamic contents. However the security flaws that we will describe in the following sections may prevent tout-court adoption of DHT-like solutions.

3.1 Sybil attack

3.1.1 At a glance

In 2001 the first Distributed Hash Tables made their appearance, stemming from the growing popularity of peer-to-peer systems and community [76, 85, 79, 103] and quickly becoming a popular research field. No more than a year after, the *Sybil attack* [24] was reported as a possible (and serious) security threat for *peer-to-peers* (P2Ps) systems in general and, obviously, for DHTs as well.

In a nutshell, a vanilla *Sybil attack* consists of one (or more) malicious entity that spawns on the attacked network a set of fake identities. These fake identities will, to some extent, behave as the legitimate ones, thus tricking other non-malicious nodes participating in the network to accept them as valid contacts (Figure 3.1 shows a graphic example of the attacker behavior during a *Sybil attack*). From another node's point of view, all other instances are required to act only in response to an incoming query: for this reason an attacker can effortlessly emulate an arbitrarily large number of network nodes: Also, attacker's simulated nodes can be strategically "positioned" in the overlay network structure by carefully choosing the identities' IDs, for example accordingly to the *distance function* used by the target system when dealing with DHTs. Phony identities used by a malicious entity performing a *Sybil attack* are often referred to as "Sybils".

A *Sybil attack* exploits the fact that in a distributed environment, like a P2P system or a DHT, the system that has no direct physical knowledge of remote participating entities. These are perceived instead only as informational abstractions or "identities". The ability to determine whether two ostensibly different remote entities (or identities) are actually operated by distinct instances is a non-trivial task.

During a *Sybil attack* the hostile entity can combine it with other attacks, either to enhance these attacks or to enhanced its *Sybil attack*. A couple of the attacks that can benefit from the *Sybil* one (and, in fact, are usually performed together) are the *Node insertion* (see Section 3.2 for details) or *Eclipse* (see Section 3.4) attacks.

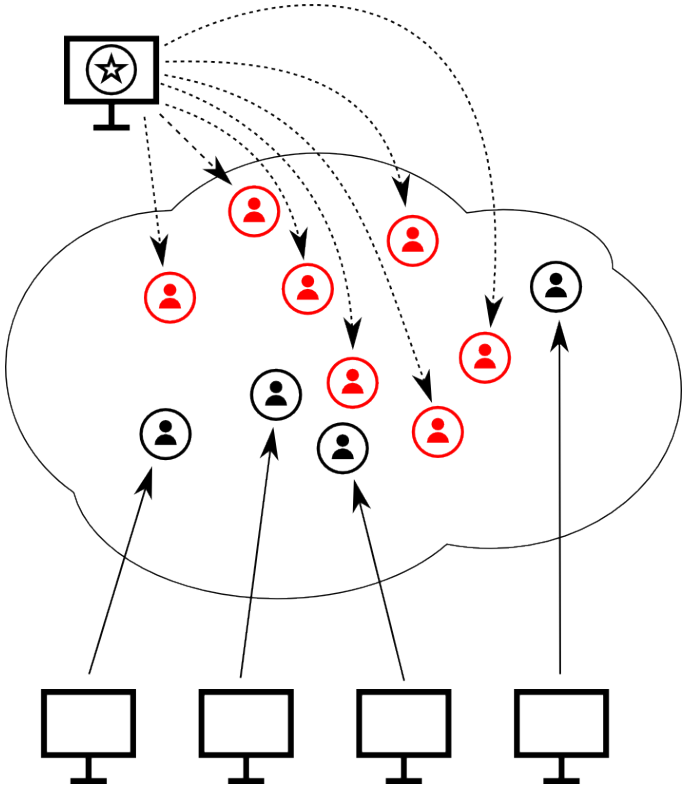


FIGURE 3.1: A *Sybil attack* example. On a network where honest nodes only deploy one ID each, an attacker (the one marked with a star) deploys several IDs (marked in red).

3.1.2 Practical relevance

Presence of Sybils in a P2P system can cause a variety of malfunctions or unwanted effects. Please note that a *Sybil attacks* are not limited to DHTs: exploiting lack of central control and nodes' self-authentication it can be performed on any decentralized P2P system. *Sybil attacks* performed on P2P systems have mainly five different targets [75]. Next paragraphs will present the possible variations of this kind of attack.

When performed on DHTs a main goal for *Sybil attacks* is the alteration or disruption of the *Decentralized Object Location and Routing* (DOLR) algorithms representing the core or the system itself. In basic DHT designs, for example, load balancing is often attained because nodes are assumed to be randomly and uniformly

distributed on the *identifiers* address space. Please note that a *Sybil attack* striking a DHT is seldom performed in its vanilla form, as already mentioned it is usually performed alongside other attacks presented in the next sections of this chapter.

Sybil attack can disrupt a P2P distributed storage application (not necessarily DHT-based). A peer-to-peer system providing this service depends on replication and duplication mechanisms to grant resources availability. Presence of Sybils can subvert system reliability: an attacker entity can gain control over the dissemination of resource copies. Thus *Sybils* can gather all data copies, modify or wipe them out; then the attacker can leave the system undetected.

Any distributed voting aggregation system is manifestly vulnerable to *Sybil attacks*. To offer a distributed voting service, a system must grant access to a collection of identities each authorized to cast a vote, choosing among (different) options. If an attacker is able to submit to the system an arbitrary large number of identities, it will be granted an accordingly large number of vote casts. Thus it is obvious that attacker is able to decide the vote outcome.

Vehicular Ad-hoc Networks (VANETs) are sensor networks allowing vehicles on the road to exchange traffic information with each other, whose popularity is currently growing. These networks can contribute to the improvement of road safety by advance warning about accidents (or other specific threats to safety) as well as leading to a more fluent traffic allowing an efficient navigation avoiding jams or temporary detours. Presence of Sybils created by a malicious entity on a VANET can mislead drivers about the current traffic situation, purposely slowing down other users or allowing the attacker to benefit from artificially cleared roads (e.g. advertising a traffic jam to drive away other VANET-equipped cars). Similarly to VANETs, the more generic *Mobile Ad-hoc Networks* (MANETs) are afflicted as well by *Sybil attacks* in a similar way.

In sensor networks query protocols are often employed rather than returning the reading of each individual sensor to prevent excessive energy consumption [60]. Exploiting such protocols *Sybil* identities may report incorrect readings thereby influencing the overall computed aggregate. Thus using enough identities a malicious user may be able to significantly alter the aggregate result.

Moreover a study has shown that in a DHT scenario Sybils summing up to 10% of the active identities are able to accumulate up to 60% of routing table entries within 48 hours from the beginning

of the attack [41].

3.1.3 Proposed solutions

Through the years many solutions have been proposed for the *Sybil attack* variants. In this section we presents the most relevant ones.

In recent years technological advancements brought an increase in popularity of MANETs and VANETs and sensor networks. So a plethora of techniques aimed at detecting Sybil nodes in these environment have been proposed [100, 49, 47, 89, 40, 1, 74, 70, 43]. Detection in such scenarios is usually performed monitoring unusual nodes' physical behavior (for example nodes' mobility in MANETs VANETs); gathering gathered through network architecture; or directly probing nodes, often using the honest nodes as probes. These techniques, tailored on features available only to sensor networks, are not very practical when dealing with a general DHT network.

Douceur in his original work about the *Sybil attack* states that these attacks are "always possible without a logically centralized authority" [24]. Following this assertion, some of the proposed solutions on DHTs are indeed based, one way or another, on *Central Authorities* (CAs). For example has been proposed an enforcement of IP addresses' authentication [41] using and external CA as a surrogate providing this service to the DHT; alternatively institution of a dedicated CA has also been proposed [14]. Eventually another proposed solution relies on a small subset of nodes self-organizing in an hierarchical "collegial" CA [78].

A popular strategy used against *Sybil attack* in DHTs is reputation/trust based [22, 3, 71, 92, 16, 12, 33, 103]. In these defense mechanism nodes are granted a certain degree of trust – i.e. reputation – typically as a reward for keeping a good behavior or for correctly serving incoming queries. Optionally these techniques incorporate some kind of randomization: when choosing the routing-path toward a desired destination a node will both take reputation into account and make random choices for each hop involved in the resulting route.

Another thriving group of techniques tackling Sybils is based on direct social links between users – e.g. an actual friendship between two users – or inferred links gathered from external social networks already available [51, 98, 97, 62, 52, 87, 37, 4, 6]. These

mechanisms rely on the trust that is embodied in existing social relationships between users that should grant the authenticity of their corresponding software nodes/identities. All social-network-based Sybil defense schemes make the assumption that, although an attacker can create arbitrary Sybil identities in social networks, he or she cannot establish an arbitrarily large number of social connections to non-Sybil nodes. As a result, Sybils are woven in a tight network component but are poorly connected to the rest of the network, especially if compared to non-Sybil nodes. Social-network-based schemes leverage this observation to estimate if an observed identifier is or is not a Sybil node.

Finally, the number of identities that a single entity can uphold may be limited by means of computational challenges. This approach was already suggested by Douceur, but he proved that direct validation works only if all identities provide proofs that are validated simultaneously (thus requiring, as already said, a CA) [24]. Since this is not feasible in a dynamic system where nodes can join and leave at any time, a computational challenge based defense require a periodic re-validation of all identities [11, 78, 53, 86].

3.1.4 Open issues

As seen in previous section, a lot of proposals to defend against *Sybil attacks* do exist. In this section we analyze the main shortcomings of such mechanisms.

First and foremost, despite the purpose, in the context of a distributed system (be it a DHT or any other kind of P2P system) usage of CAs is not advisable because it will create a single point of failure. Creating one would be unreasonable since a distributed system is (also) devised to avoid such a criticality and, from a more philosophical point of view, essentially betray the spirit of this kind of systems.

All the *Sybil* defense schemes relying on social links (or trust gained through social links) make the assumption that *Sybils* can only form a certain number of links to non-*Sybil* nodes. However, it still is an open question whether this is true in any online social network [91]. It has been shown that, at least in some social networks, this assumption does not hold being identity theft possible [10] thus frustrating the mechanisms belonging to this category of defenses.

Proposed solutions addressing MANET or VANET networks relies on physical behavior or on data specific to sensor networks. Thus the majority of these countermeasures cannot be ported to more generic distributed systems like P2P systems or DHTs, making them irrelevant for the scope of this dissertation.

Finally computational challenges appear to be a good and simple strategy against *Sybils*. However, as pointed out in Douceur's the original work [24], a straightforward implementation based on single validation is proved to need a CA. Thus for this technique to be successful is required an additional mechanism enforcing a mandatory periodical renewal of proof-of-work authenticating a node. Granted the existence of this extra component, they appear to be the most effective strategy now available.

3.2 Node insertion

3.2.1 At a glance

A particularly mischievous kind of attack that DHTs are vulnerable to is the *Node insertion attack* [83, 88, 55, 50], described for the first time in 2002. *Node insertion attack* is performed by a malicious node that could join and participate in the look-up protocol correctly. When asked however it deny the existence of data it is responsible for. Alternatively, it might claim to actually store data when asked, but then refuse to serve it to clients. Also, it could serve bogus routing information during exchanges (correctly carried out) with other nodes. *Node insertion attack* can be referred to with the alternative names of *Routing attack* or *Storage attack*.

To specifically hide a particular resource performing a *Node insertion attack*, the attacker needs to strategically position one or more node IDs. These IDs are selected according to the DHT's *distance function* in a way that makes of them the most suitable resource's owner identifiers. Once the attacker has secured the desired IDs it may, for example, accept *store* requests thus convincing honest nodes that it successfully saved the record. Up to this point the attacker behavior sticks to the normal DHT protocol. When queried for the stored values though, the attacker hiding behind the phony ID will deviate from the standard protocol and maliciously reply with bogus data or just denying the existence of the record.

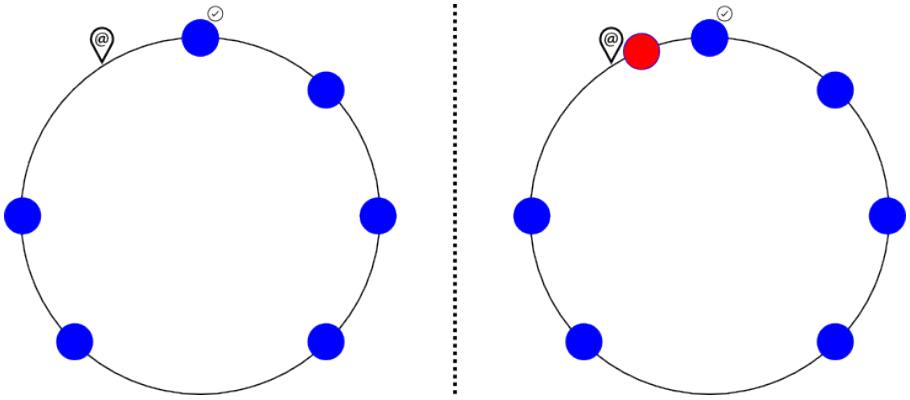


Figure 3.2: A *Node insertion attack* example. In the left part of the image a Chord-like network is shown (see Section 2.3.2 for a reference on Chord mechanics). The placeholder marked with an @ symbol represent the target resource's ID position. The rightful node responsible for the resource is one the blue one, marked with a small check sign upon it. In the right part of the image is depicted the same network after a malicious node (the red one) perform a *Node insertion attack*, purposely becoming the closest node to target resource. All *look-up* queries for the resources will be directed toward the attacker node after its inception.

An attacker performing a *Node insertion attack* can boost its effectiveness by jointly performing a *Sybil attack* (see Section 3.1). Incrementing the apparent number of attackers node can ease resource's takeover and facilitate to overpower a DHT's redundance system (if any).

3.2.2 Practical relevance

Practical relevance of a *Node insertion attack* should be manifest since it permits an attacker to suppress or alter records stored in a DHT. Many example for this kind of attack can be provided, next paragraphs will describe some possible *Node insertion attacks*. Moreover last paragraph will provide some data gathered through attack experiments performed on an actual DHT network.

Since DHTs are a critical layer of many decentralized or P2P network systems, this kind of attack represents, in many ways, a strong shortcoming for information dissemination. Many practical

examples are represented by actions taken on those P2P file-sharing services where pirate copies of copyrighted media are exchanged: in such a system a company owning licenses and royalties about shared material can perform a *Node insertion attack*, thus preventing the spreading of particular file names that lead to material whose rights are managed by the company itself. In this case the attack is considered to be an ethical action [99].

Another example, this time of unethical nature, can be a system hosting distributed websites and microblogs or allowing direct P2P communication (in the fashion of Osiris or RetroShare, see Section 2.3.5.1). This kind of systems could be a useful tool for real-life dissidents and whistle-blowers – e.g. trying to report abusive treatments by oppressive government, ecc. A censorship authority could then perform a *Node insertion attack* and easily make unavailable those services, effectively silencing any opinion standing out of line.

Node insertion attack affects data availability, thus its effects and consequences are easily understandable. An interesting work performed onto the Kademlia based KAD network (see Section 2.3.5 and Section 2.3.5.1 for details on Kademlia DHT design and on its implementations) shown that, in a real-world large scale active DHT, this kind of attack can hijack up to 95% of the *look-up* requests colluding as little as 3 attacker IDs [55]. The necessity for at least three malicious IDs, differently from the example provided in Figure 3.2, is due to the replication factor used by KAD. Elaborating on the attack described in [55] a more efficient one was devised and performed, thus improving the success rate by a factor of 3.8 times [50].

3.2.3 Proposed solutions

Two main strategies are used in the effort of protecting a Distributed Hash Table against *Node insertion attack*. These relies on redundant routing and redundant storage [88].

Solution proposals belonging to the class of data redundancy are based on erasure coding techniques [21, 28, 65]. Coming at the price of potentially higher latencies and a higher system complexity, erasure coding offers less storage requirements compared to simple data replication. On the other hand plain data replication enables more simple algorithms for maintenance and verification. There are several approaches to data replication, ranging from storing

replicas in nodes close to one another in the identifier space [79, 85, 63, 14, 34, 28, 68, 95], to store them at equally spaced locations over the identifier space [32] or on random locations across the address space [102]. Another method consisted in combining said approaches, selecting locations spreaded over the identifier space and storing replicas in nodes close to each of these locations [8, 27].

Redundant routing is necessary in order to reliably locate nodes responsible for a given key (both in a redundant and non-redundant storage scenario). There are two approach that implement redundant routing: multiple paths [14, 32] and wide paths [63, 34, 68, 28]. Wide paths are suitable when replicas are stored at nodes close in terms of the DHT's *distance function*. Multiple paths instead are a better match when replicas are spreaded over the identifier space. These two approaches can be combined as well – becoming multiple wide paths – by trying wide paths successively. Anyway this has the disadvantage that maintaining consistency when replicas are mutable entries may become cumbersome and expensive in most DHT designs. A concrete example is the approach adopted by Myrmic [95]: it is similar to wide paths but instead of trying multiple nodes simultaneously at each step, it tries one node and resort to an alternative one only in case of failure.

Please note that some DHT improvement proposals moreover do not make an explicit effort against the *Node insertion attack* considering it just a byproduct of other attacks, like the *Sybil attack* (see Section 3.1). An example of this is *Persea* DHT [5]. It has some kind of storage redundancy in place, relies on the assumption that its network is free from *Sybils* and thus if a key is unavailable at one location, it will be still available from another location that should not be controlled by malicious nodes.

3.2.4 Open issues

Although a number a strategies has been proposed to cope with the problem of *Node insertion attacks*, there are still open issues about a reliable solution. This section will briefly describe how a *Node insertion attack* is still possible even in a DHT where countermeasures Section 3.2.3 are put in place.

About redundant storage a common strategy attains data replication through a local (in terms of the address space) cluster of nodes. In such a case a small number of malicious nodes can con-

centrate, possibly also performing a *Sybil attack*, on the overlay's specific region and take control of all replicas. Nonetheless, spreading replicas over the identifier space though is not necessarily a big improvement: since the placement algorithm must be publicly known to coordinate honest instances, malicious nodes may attack all the relevant location for a target resource with a small additional effort.

Data redundancy moreover is not enough to prevent storage attacks: the network must provide reasonable guarantees that nodes are not able to select their own location in the identifier space. The most straightforward way to achieve this is the use of random identifiers issued by a trusted certification authority able to limit the fraction of malicious nodes [14]. Alternatively is possible to have a set of nodes generating identifiers using a Byzantine-fault-tolerant consensus algorithm to assign the ID to a new coming node; on top of this is artificial churn is induced in the network to prevent concentration of malicious nodes at specific regions [18, 8]. However this does not limit the number of attackers: thus these countermeasures should be coupled with other techniques suitable to limit *Sybil attacks* (see Section 3.1).

Is known that usage of erasure coding provide storage savings. However the bandwidth required, in particular under a varying degree of churn, to maintain appropriate redundancy levels is approximately the same for both coding and plain replication [77].

As a final note, claiming that a system free from *Sybils* is also protected against *Node insertion attacks* is rather naive. As already stated in Section 3.2.2, in real-world applications just three attacker nodes can be enough to subvert the whole system [55]. This means that, although a coordinated *Sybil attack* can make easier to perform a *Node insertion attack*, it can as well be performed by a collusion of a small number of physical attackers, not necessarily spawning multiple phony identities.

3.3 Publish attack

3.3.1 At a glance

A different kind of attack that, like the *Node insertion attack* (see Section 3.2), aims to disrupt DHT's data availability is the

Publish attack [55, 38, 30]. *Publish attack* – also called *Index poisoning attack* or *Index pollution attack* – exploit the fact that, for practical reasons, DHT implementations do store record index tables of limited capacity. The attacker exploits this feature trying to publish, by means of correct and "legal" *store* procedure, large amounts of information in multiple entries. Stored data can be altered versions of target records or just bogus data.

During a *Publish attack*, depending on which *store* strategy was chosen by the attacker, two different outcomes are expected to happen. Either attacked peer will not accept any other *store* once its index tables are full or, alternatively, as a result attacked peers will only return attacker's poisoned entries instead of the correct information.

Compared to other attacks, and in particular to a *Node insertion attack*, a *Publish attack* requires fewer skills to set up a convenient software and, by far, less computational resources.

Publish attack does not only affects DHTs but can target any P2P system relying on indexes stored at peer nodes whose data may be submitted by other nodes.

3.3.2 Practical relevance

A *Publish attack* is a very simple kind of exploit to which DHTs are vulnerable and, both on Mainline and KAD networks (see Section 2.3.5.1), it has been studied and measured [55, 46, 54, 101]. This section describes the practical uses of this attack along with experimental results available in literature.

Publish attack has been studied and used both for ethical purposes other than as an attack strategy. Like in the case of *Node insertion attack* (see Section 3.2.2) there exist a so called ethical usage of the *Publish attack*, consisting in carrying out index poisoning to inhibit distribution and download of copyrighted materials. The "copyright industry" has an understandable and significant desire to prevent unauthorized distribution of content through P2P systems, thus avoiding huge financial losses [99]. In fact alongside the research interest trying to make *Publish attacks* more efficient [69, 73], to prevent copyright infractions the industry is known to hire specialized companies performing this attack, each with its own decoy techniques and strategies [54]. On the other hand, used as an

attack strategy *index poisoning* is an easy, cost-effective tool used to maim P2P system's data integrity.

It has been reported that *Publish attack* works fairly well on average, achieving a success rate of roughly 80%. Although it is lower than the success rate of a *Node insertion attack* (see Section 3.2.2), it is important to note that to perform an *index poisoning* a single attacker node is required thus making a *Node insertion* a much more expensive alternative [55]. Moreover, in 2011, it has been measured that – on the KAD network – about 20% of audio file entries and about 41% of video file entries are poisoned [101]. Luckily due to permanent arrivals of new peers and departure of existing ones, the success rate periodically drops and remains low for a certain time period before it recovers, due to malicious nodes re-attacking the peers periodically [55].

As a final consideration note that some DHT systems – e.g. P2P file sharing – as value associated to a key store a reference to the network addresses of those nodes that are supposed to own a copy of the searched resource. As a result the common expected behavior after references' retrieval is for the searching node to directly contact the discovered hosts. Instead of inhibiting access to a particular resource, an attacker can use a *Publish attack* to poison various popular DHT keys: thus the attack can be tuned to cause a powerful *Distributed Denial of Service* (DDoS) attack against arbitrary unsuspecting victims [54, 99].

3.3.3 Proposed solutions

Countermeasures to *Publish attack* can be divided into two categories: proactive and reactive techniques [99]. This section will outline both.

A proactive technique proposed for Mainline DHT is as simple as crosschecking IDs in communication or keep-alive messages with those stored in the local routing table. This forces a possible attacker to maintain the state information for each previously contacted node, also increasing its computational overhead. This is expected to result in a mitigation of possible *Publish attacks* [94]. Another technique relies on measurement of advertised DHT records over a certain period of time. These checks helps in the determination of which ones are poisoned, polluted or clean. Basing on those observations a node could be able to estimate – also beforehand –

if a record *store* is legitimate or not [54]. DDoS attacks induced by a *Publish attack* can also be tackled proactively: using an *Identity Based Signature* (IBS) scheme is possible to protect a P2P network from modified and forged indexes. This is done by tracing any advertised record to its publishing peer [58].

The reactive methods used to prevent *Publish attacks* include the followings: blacklisting; reputation and voting schemes; and the collaborative filtering and pollution modeling [61, 99, 38, 19]. These strategies are conceptually easy to understand. Honest nodes can try to authenticate versions and record advertisements; alongside this feature, the system may allow users to actively rate sources and helping in the prevention of the attack [54]. In any of the reactive schemes, as soon as a peer performing a *Publish attack* is identified it gets blacklisted, thus cutting out its ability to tamper records.

Although *Publish attack* is common in real-world DHTs (as stated in Section 3.3.2), it countermeasures are fairly effective and – especially considering the reactive techniques – they are often relatively easy to implement. Note that when a system undergo only a light *Publish attack* poisoning, the reactive methods are more efficient due to relatively lower resources consumption.

3.3.4 Open issues

Blacklisting malicious nodes to avoid further *Publish attack* activity is an easy albeit effective countermeasure. Note that if based on reputation systems, it has the shortcoming that new peers joining the network may not have any reputation based on previous rating: thus it can experience an unjustifiable (maybe temporary) exclusion from the system. Moreover the usage of a reputation system heavily depends on users reliability: if an (even small) fraction of malicious users gives a wrong and/or false feedback the global performance will suffer a significant decay in its effectiveness.

Instead considering the IBS mechanism proposed to defend against poison-induced DDoS attacks, the success of such a protocol requires the usage of a *Private Key Generator* (PKG) to be registered with a trusted certification *Central Authority* (CA). Such a CA than could represent a single point of failure and, as already discussed in Section 3.1.4, can be considered unacceptable.

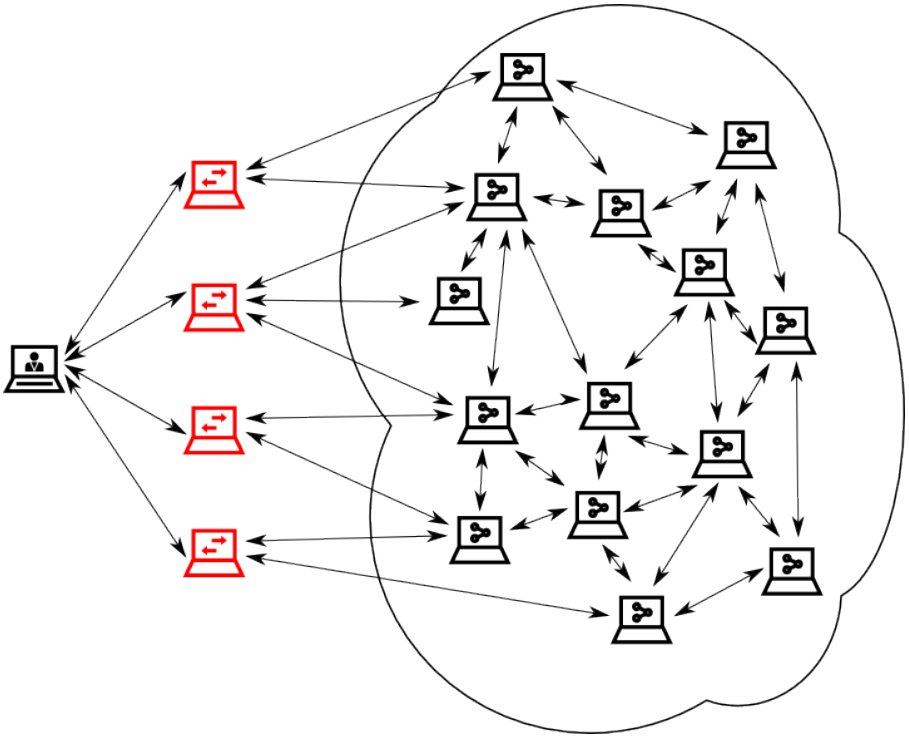


FIGURE 3.3: An *Eclipse attack* example. The right part of the image shows a generic DHT network. The node on the extreme left part of the image wanted to join the DHT but it ended up connecting only to four malicious nodes (marked in red). Thus all victim node's connections to the main DHT network are carried out throughout the set of malicious nodes that are able, if and when they desire, to isolate the victim from the rest of the network.

3.4 Eclipse attack

3.4.1 At a glance

When performing an *Eclipse attack* the attacker tries to partition from the main network a node or a set of nodes [55, 30, 88, 38]. To attain this goal an attacker needs to gain control over a sufficient number of the neighbors of the target node(s). An alternative name used to denote *Eclipse attack* is *Routing Table Poisoning attack*, since it based on inserting poisoned entries in the victim's routing table.

An honest node is vulnerable to this attack especially during its *bootstrap* phase: when a node *bootstraps* may need to ask for new nodes contact. During this phase attacker-controlled neighbors may be fed to *boostrapping* node, thus filling its *routing table* with malicious nodes' contacts. Another way to poison a node's routing table in an attempt to separate it from the network can be carried out is during its "normal" communications: once at least one of the nodes controlled by the attacker is inserted in a target node's routing table, the victim node will from time to time contact the attacker-controlled node during normal routing operations. When this should happen, each message exchanging routing data updates will be filled only with malicious contacts. Target node will then use this poisoned data to update its own routing table.

A graphical depiction of a node "eclipsed" from the main network is shown in Figure 3.3.

3.4.2 Practical relevance

Instead of trying to poison *look-up* outcomes like in the *Node insertion attack* (see Section 3.2), an *Eclipse attack* aims to gain control over the routing mechanism of a P2P system [55, 30, 96]. Compromised nodes will work together and try to fool any genuine node by adding their addresses to the genuine nodes' neighbor list. Once an attacker gains control over the routing mechanisms, nodes cannot correctly forward a message to desired destination. This way attacker-controlled nodes may also be used to put in place a man-in-the-middle scheme, allowing to control or alter messages conveyed through the malicious nodes.

Eclipse attack can be used to facilitate other attacks like *Node insertion attack* and, on the other hand, can be boosted with a preventive *Sybil attack* (see Section 3.1). Creating a number of *Sybils* to be injected in target's routing table, an attacker can substantially reduce the amount of resources actually needed to successfully *eclipse* a node.

Unstructured overlays are more susceptible to this type of attacks than the structured ones: within a structured environment constraints over the neighborhood of one node are naturally imposed. For this reason, unstructured overlays usually relies on floods mechanisms to collect information about topology of the network. The more nodes use these mechanisms, the higher the probability

that an attacker will gain control on more nodes in the system: with every update retrieved during a flood the overall number of sources of malicious entries contacted increase [30].

On the KAD network (see Section 2.3.5.1 for details on this Kademlia implementation) it has been shown that, during performed experiments, *Eclipse attack* success rate reached virtually always 100% within minutes, although experiments was limited to target merely single victim peer [55].

3.4.3 Proposed solutions

In this section we review proposed countermeasures against the *Eclipse attack* reported in literature. Against this class of attacks to consider a defense successful it is expected to roughly bound the fraction of malicious entries (inserted in honest nodes' routing tables) to the fraction of malicious nodes present in the whole system [88].

A possible defense strategy relies on the use of two concurrent and distinct routing tables [14]. One table – namely the *optimized table* – is continuously updated while operating the system, thus being exposed to potential corruption due to malicious nodes' activity. The other table instead must contain only verified entries. A node decide which routing table should be used to route its messages, basing it decision on routing failure tests. This strategy can be further enhanced by periodically resetting the *optimized table* to the one with trusted contacts, thus purging any possible malicious entries contained [18].

Another technique to provide some kind of resistance against *Eclipse attacks* resides on artificially induce churn in the network. Whenever a node joins the system, all the other nodes that become its neighbors after its inception leave the network; these nodes then re-join the system with a new random identifier immediately [8].

In systems where the overlay network is more structured a basic form of defense is to impose constrains on the nodes' identifiers allowed in routing tables; this strategy was already suggested in the original work presenting Chord DHT [85] (also see Section 2.3.2 for basic details on Chord). Moreover forcing nodes to select their identifier through a *Certification Service* (CS) should mitigate the attack (since freely choosing its identifier may help an attacker to reach a target node's neighborhood) [59].

Another popular countermeasure is based on the fact that during an *Eclipse attack* malicious nodes' connections degree tend to be higher than the average degree of honest nodes in the overlay. Leveraging this feature an honest node can select its neighbor among those with a degree below a certain threshold; thus the probability of selecting honest nodes increases and the network is expected to gain protection against the attack [82, 38, 96].

3.4.4 Open issues

Despite a number of defenses against *Eclipse attack* have been proposed, such an attack still remains an open problem [82]. Each and every technique described in Section 3.4.3 sports some serious drawbacks. This section will describe them.

The techniques based on the usage of two different routing tables (the *optimized* one and the trusted one) tend to cause more overhead than advantages. After some time the system is suffering from an *Eclipse attack* most of the routing performed ends up to be done using the verified routing table, with an additional overhead due to routing attempts via the *optimized* (and already poisoned) routing table [18]. Moreover periodically resetting the *optimized* table proves useful only if tables' poisoning increases slowly over time [88].

Adoption of induced-churn strategies, although providing some form of protection against the *Eclipse attack*, leaves the system vulnerable to other exploits. Purposely inserting new nodes in the overlay, an attacker can trigger extremely high churn maiming the system. Moreover an attacker may remove from the overlay a number of honest nodes by means of a *Denial of Service* (DoS) attack while other nodes perform the leave and re-join phase [88].

Using a *Certification Service* (CS) may be troublesome to implement eventually falling back to the usage of a *Central Authority* (CA). That, obviously, would become a single point of failure for the whole system and is not acceptable as we previously discussed it in Section 3.1.4.

Finally, bounding the nodes' connection degree, although completely decentralized, tends to decrease system efficiency. Requires a small degree per active node results in an increased *look-up* time even in the absence of attacks: in general the less contacts are available, the more routing hops are required.

Our new secure DHT 4

In this chapter we present or main contribution: two new DHT primitives. We will describe how they can constitute a line of defense against attacks aimed to selectively obscure or alter resources – for a disquisition about the security threats affecting Distributed Hash Tables please refer to Chapter 3. We will also explain how these new primitives can be integrated –requiring minimal adjustments – with the a basic DHT design, enhancing it. In next Chapter 5 we discuss correctness of our enhanced design.

To avoid attacks, especially *Node insertion* and *Sybil attacks* (respectively see Section 3.2 and Section 3.1), the set of primitives available to a basic DHT design described in section 2.2 must be extended. The required ingredients are three: randomness, age and work. These three ingredients can be granted by just two new primitives: *getSeed* and *isOldWorker*. To integrate these new features with the basic design, naturally, does require to adapt the DHT model but, as we will see, this does not affect the conceptual mechanics habitual to any DHT. In the next paragraphs we will individually explain the behavior of the new primitives and, later, we detail the adjustments required for integration.

The *getSeed* primitive can be used by a node at any time enabling it to retrieve a "random seed" – namely "random seed" is a random byte string. Data returned by *getSeed* must have three properties: it must be a network-wise consistent value; it must change every T time units; and, obviously, it can not be guessed or foreseen

before the random seed itself is actually generated. These three properties also imply that as time passes *getSeed* creates a sequence of random seeds consistent across the network. Also note that, as all DHT nodes can (and will) access to the sequence of random seeds repeatedly calling on *getSeed*, the evolution of the value returned by this primitive implicitly creates a time framing. This seed-based time framing can be used to timestamp – in some sense – the events occurring within the DHT network. This primitive provides the "randomness" aspect of our enhanced DHT.

A proper timestamp is provided by *isOldWorker*. Upon joining the network each participating node chooses its own *node ID*. In our model we do not impose any constrain on *ID*. However it is advisable – for honest nodes – to randomly chose its own *ID*: this eases stored keys load balancing for the whole network. Clearly inception of an *ID* (and it relative node) happens during the an "epoch" marked by a random seed: the seed returned by *getSeed* up to the inception moment. Therefore *isOldWorker* allows any node to perform a check on another node *ID* inception age. More precisely this primitive returns *true* a certain *ID* has been choose by a node during a seed epoch of a *previous* seed (relatively to the one currently returned by *getSeed*). Moreover, for reasons later explained, we ask that the node inception is not "too old" – say no more than η seeds old. Otherwise – i.e. there is no evidence of the *ID* being incepted in a previous but reasonably recent epoch – *false* is returned. This part of *isOldWorker* behavior is responsible for the "age" feature.

Regarding the "work" facet of our new design *isOldWorker* is again the key primitive. Accessing this primitive a DHT participant can verify that inception of a certain *ID* required submission of a (valid) proof of work [39] to the node that spawned it. This proof of work must be tied to the specific *ID* as well as to its inception's seed. This certifies that the "right of use" to an *ID* has been "payed" by the node claiming such an *ID*, thus mitigating *Sybils* outbreak. Any protocol providing the function of a proof of work system is acceptable for this stage. However the general nature of a DHT is that of a network system: therefore an optimal proof or work scheme would undeniably be a network-bounded one – e.g. a puzzle similar to an Abliz's guided tour [2] but heavily less structured and centralized – rather than a (more traditional) CPU-bounded one like, for example, Hashcash [9]. As we said, when checking age of a

node, we must also verify that it is not "too old". Otherwise waiting an indefinite amount of seed epochs, an attacker would collect a considerable amount of valid proof of works, for a large number of *IDs*. So the constrain about being at most η seeds old is needed to keep the proof of work scheme valid. Please note that *isOldWorker* requires that *both* the constrains on the *ID* are satisfied: it must be "old" enough and, at the same time, at *ID*-generation time a proof of work was payed. If any of these two conditions is not met, the primitive will return *false*.

Thus far we described how each of the new primitives individually work. Now we will proceed detailing how they are integrated on a general basic DHT design, leading to an enhanced DHT design. We will begin by showing how local operations performed by every node must be modified to comply with our new setup. Then we will proceed explaining how the new primitives will work together when nodes of the enhanced DHT interact with each other.

First of all *getSeed* primitive will affect resource positioning within the DHT. Our new design, as the basic one, has its cornerstone on distributing key/value resources according the hashed value of their keys. However in our enhanced DHT whenever a node calls the *hash* function to convert a resource's key to its *ID* also *getSeed* must be used. More precisely, instead of simply hashing the key value, we require that a resource's *ID* must be obtained by hashing together both the key and the random seed returned by *getSeed*. Thus the same resource will have different *IDs*, changing any time *getSeed* output – i.e. the seed epoch – changes. A resource's *ID* also marks the resource's position within the DHT address space. Note that the sequence of *IDs* assigned to a resource during different epochs represent the address space "trajectory" followed by the same resource.

Nodes of an enhanced DHT must use contacts accordingly to *isOldWorker*. For any new node reference obtained during any of the normal Distributed Hash Table's operations, a check through *isOldWorker* is performed. This new reference can be inserted into the *routing table* of a node if and only if *isOldWorker* confirms that new refereced node's *ID* was chosen before the current random seed returned by *getSeed* was generated. This guarantees that all nodes' addresses and *IDs* stored in a *routing table* was incepted in a previous random seed "epoch". Note that enforcing *isOldWorker* control over the *routing table* also affects the outcome of *look-up*

and *store* procedures. Also note that an extra care must be put in *which* nodes get actually inserted in a node's *routing table*: this is due to a certain degree of *routing table poisoning* risk. This last issue is addressed more directly in Section 5.4, please refer to it for further details.

Finally network communications can also be affected by the *isOldWorker* proof of work token. If a node doe not proves to be old enough, it can still be allowed to a limited number of DHT operations – those not requiring any special trust like, for example, simple *look-ups*. A node will drop any communication with any node that, failing *isOldWorker* due to its fresh inception, does not at least provide a correct proof of work – i.e. the correspondent node is not providing a proof of work valid for the current time frame[1]. As already mentioned, a node is expected to invest a certain amount of work to be entitled to the usage of its *node ID*. Failing to provide evidence of such an investment results in the node being ignored from the rest of the network.

These are the slight modifications required to nodes' behavior. We will now point out how such small changes reflect on the global DHT functioning.

As explained dealing with the usage of *getSeed* coupled with resources' hashing, it should be clear that usage of *getSeed* as cornerstone for the *hash* function – alongside *getSeed* output evolution – does imply that during each random seed "epoch" a resource will have an *ID* different from the previous ones. We will call such an event – resources changing *IDs* when *getSeed* changes – *resources rotation*. Note that *getSeed* output is random and a Distributed Hash Table's *hash* must be a function hard to invert. Thus during a *resource rotation* the "destination" *ID* of a resource is not predictable before the *rotation* actually takes place.

A secondary effect of *resources rotation* is a natural management of stored records' expiration: when a seed epoch changes a certain resource will be entrusted to a new node. This node however does not necessarily know, yet, the existence of a resource previously stored elsewhere. Then it is responsibility of the node that originally stored a record to renew it by issuing a new *store* request, just as it would do upon a "normal" resource expiration.

[1]Remember that the succession of different random seeds returned by *getSeed* do imply a time slicing.

Any node in the network enforces a *routing table* insertion policy ruled by *isOldWorker* primitive and, if necessary, a communication drop policy managed by this primitive proof of work. These policies grants to any node that any other valid DHT node observed both chose its *ID* before last *resources rotation* happened and granted its *ID* by performing some work. Furthermore as we previously hinted any of the basic, yet modified, DHT primitives will lead to "old" nodes. This is due to the fact any reference stored in a *routing table* is granted via *isOldWorker* primitive. Thus resources will be given in custody – i.e. *stored* – only on nodes that are both "old" and actively maintaining their *ID*.

In our new DHT architecture then resources are expected to

1. be close, according to the *distance function*, to an "old" node that chose its *ID before* possibly knowing the ending position of the resource

2. *rotate* every T time units.

This ensure the DHT against *Node insertion attacks* and, more in general, against tampering on selected resources. To harm a certain resource an attacker must first become the entrusted store location of it. But no node will ever be entrusted with a resource if does not pass an *isOldWorker* check. This implies that any node actually storing resources chose its *ID* regardless of *which* resources would have been put into its custody. Of course this does not mean that a node could join the DHT and tamper with resources bestowed on it, but please note that this is not and aimed strike: randomly taking down DHT nodes – e.g. via a DDoS attack – would have an analogous impact on data availability. By chance an attacker could however become the node entrusted with the target resource. In this case however malicious node will not be able to harm the DHT for significantly more than T, the time expected before a new *resource rotation* occurs.

The new DHT primitive *isOldWorker* also provides additional security causing *Sybils* mitigation. Being a form of proof of work, presence of this additional primitive discourages usage of multiple *IDs*. On the other hand a malicious node could choose more *IDs* in an attempt to increase the probability of becoming, by chance, the storage location for its targeted resource. This would require a considerable effort compared to the attack effectiveness. Also, due

to proof of work expiry, *Sybil IDs* generation is bounded by the adversarial capacity of generating them.

Concluding, we propose a Distributed Hash Table with an extended set of primitives. In basic DHTs an attacker can effortlessly corrupt or delete resources. With few adjustment a DHT can be turned in a system with increased security. In such a system although does not completely eliminate threats, attacks are remarkably harder to be carried out.

Security guarantees 5

In this section we formally prove that any DHT satisfying some very mild and "standard" conditions can be protected from a large range of attacks by the use of the two primitives $isOldWorker$ and $getSeed$, even if those attacks are carried out by an adversary that controls a substantial portion of the network's computational power. We explicitly remark that we do not attempt to prevent or obviate brute-force *Denial of Service* (DoS) attacks aimed at disconnecting one or more nodes simply by directing against them excessive traffic; there are simple techniques based on virtual node replication that can obviate these attacks, which are beyond the scope of this dissertation.

5.1 Preliminaries

A first distinction we make is between *active* and *passive* nodes. The former are those that *support* the DHT, and in particular that can be part of each node's routing tables (and be returned by *look-up* queries) and on which one may *store* information. The latter are all other nodes, that can *use* the DHT, but do not support it; in particular, passive nodes can search and store information on the DHT. Simply put, any node that wants to take an *active* role when interacting with other nodes must pass an $isOldWorker$ check; otherwise it is treated as a passive node.

We focus this discussion on active nodes, noting that the only obstacle that passive nodes may present to the correct functioning of the DHT is by requesting "too much" service. This is easily obviated by having active nodes, upon a shortage of resources, ask for progressively more onerous proofs of work, or other forms of payment, from passive nodes, until an equilibrium is reached. Since exactly how much traffic an active node can support can depend on a large number of factors, and the DHT is likely to be just a component of a distributed service that faces on its whole similar resource-limit issues (where the exact resources and limits, however, are likely to vary widely with the actual service involved) we do not explicitly consider the exact form this access control can take.

5.2 Assumptions

We make a number of simplifying assumptions in this analysis. The first is that all (active) nodes have the same computational power (more powerful nodes can simply simulate multiple less powerful nodes) and that this power is such that a node can have at most w proofs of work produced by $isOldWorker$ valid at any given time. We can then reason in terms of IDs, instead of nodes, noting that an adversary that controls a fraction f of all the computational power of the (active) network can effectively masquerade as no more than a fraction $\frac{f \cdot w}{f \cdot w + (1-f)} < f \cdot w$ of all the (active) network IDs. We shall then talk of "bad" IDs to denote those controlled by the adversary, and "good" IDs to denote all others.

Note that w is, in general, freely tunable, but there is a trade-off: large values of w grant comparatively more power to malicious nodes, while small values of w force even honest nodes to waste a substantial fraction of their computational power to prove their honesty.

The second assumption we make is that the address space is sufficiently large that we can reason about it as if it were continuous. Also, we assume that the address space into $k \geq 2$ *level* 0 *areas*, each with the same "size", in the sense that a random address is equally likely to be part of each; and the area to which the node belongs is recursively partitioned into β level 1 subareas, and so on, with each level i subarea being partitioned into β level $i+1$ subareas. We call β the *branching factor* of the DHT. Note that

many DHTs have a branching factor of 2, although some – e.g. Pastry and Tapestry – can admit branching factors that are as high as n^{ε} for some small ε.

We say that:

Definition 1 *Given three IDs x, y, z, we say that x is closer to y than to z if, for some i, x belongs to the same level i subarea of y, but not to that of z.*

For brevity, we shall hereafter use the following:

Definition 2 *A given point in the address space is* compromised *if it there exists no good ID closer to it than every bad ID.*

Finally, we assume that a new node *bootstrap*ped into the DHT can begin with knowledge of at least one good ID. Note that this is obviously necessary to avoid having the new node completely isolated from the rest of the network by bad IDs.

5.3 Proximity guarantees

Informally, if we first place the good IDs sufficiently "uniformly", no matter how we position the bad IDs, a random point in the address space will have roughly the same probability of being closest to a good ID and to a bad ID. Then, if we hash the position of a resource to a random position through *getSeed after* the active IDs have been placed, it will have roughly even chances of being "controlled" by a bad ID.

In fact, a smaller fraction of bad IDs yields even better guarantees, even though the probability of one of them being closest to the resources decreases slightly sublinearly with the fraction of bad IDs if good IDs are placed at random (informally, because the random placement will inevitably leave a few larger-than-average "holes"). More formally, we can prove the following:

Lemma 1 *Assume all good IDs are chosen uniformly and independently at random in the address space. Then, for any (subsequent) choice of bad IDs, and any randomly chosen point x in the address space, if bad IDs are at most a fraction q of all IDs, the probability that x is compromised is $O(q \log(1/q))$.*

Proof 1 *The proof is immediate from the fact that, with n nodes, the probability that any contiguous fraction s of the address space is unoccupied by any good ID is* $e^{-\Theta(ns)}$*). Thus, positioning the bad IDs in the largest such portions (which yields a total probability of one of them being closest to the random ID no larger than the sum of the fractions), one can apportion on average at most a fraction* $O(\log(1/q)/n)$ *of the address space to each bad ID, and thus apportion to bad nodes a fraction at most* $O(q \log(1/q))$ *of the address space.*

Remark 1 *Note that as long as the probability that a resource is "controlled" by a bad ID is less than* $\frac{1}{2}$*, we can easily boost the probability of making the resource "safe" by replicating it to m random positions, which makes the probability that more than a minority of them will be compromised* $O(2^{m/2})$ *– i.e. exponentially small in the replication factor.*

5.4 Poisoning the routing tables

It would be tempting to assume that the result of Lemma 1 by itself ensures that our "randomly moving resources" scheme, plus possibly an $m-$fold replication of each resource, makes the probability of an insertion attack being successful on any given epoch exponentially small in m even with a constant fraction of bad IDs. Unfortunately, this fails to take into account that colluding bad IDs can "poison" the routing tables, in such a way that even if a good ID is closest to a resource – i.e. if the resource is not compromised – all but a vanishingly small fraction of searches will fail to reach it.

To see how this is the case, recall that searching in a DHT of n active IDs proceeds, informally, in $log_\beta(n) = \log(n)/\log(\beta)$ steps; at each step the searching node "zooms" into a portion of the address space β times smaller by performing one or more *look-up*s from IDs of a given subarea, that return a number k of IDs from the next-level subarea. If we assume that each good ID returns a fraction q of good IDs and a fraction $1-q$ of bad IDs, whereas each bad ID returns only bad IDs, the fraction of good IDs available to a searching node after t look-ups is at most q^t – which becomes vanishingly small at the end of the search (i.e. for $t = \Theta(\log_\beta(n))$) if $1 - q = \omega(\frac{1}{\log_\beta(n)})$. In other words, if the branching factor β

equals 2, we can tolerate at best a fraction of bad IDs inversely proportional to the logarithm of the number of active IDs, whereas if $\beta = n^{\epsilon}$, we can tolerate a small, constant fraction proportional to ϵ. A larger fraction of bad IDs can effectively "trap" with high probability any search into *look-up*s returning solely bad IDs.

The situation is even worse, however. The crucial point is that DHT fill (and refill) their routing tables through *look-ups* to random points of the address space. This means that if the fraction of good IDs in the routing tables is at most q at a given point in time, then if the routing tables are refreshed "naively" through random *look-ups*, the next crop of IDs in the routing tables will hold only a fraction at most $\approx q^{\log_{\beta}(n)}$ of good IDs; and eventually, all entries in the routing tables will be, and remain, bad IDs.

Fortunately, we can obviate this problem, and ensure that if the fraction of bad IDs is q, and thus the compromised fraction of the address space is $p = O(q \log(1/q))$, the fraction of bad IDs in the routing tables remains $O(p)$. The key idea is to attempt multiple independent searches for the same point in the address space, and place into the routing table only the closest ID returned by any of them. If the number of independent searches is sufficiently large, one can make the probability that *all* of them get trapped by bad IDs arbitrarily small, and thus the probability of not obtaining a good ID only marginally higher than the "natural" probability that the target of the search being compromised. In particular, if we assume that a fraction at most $2p$ of the routing table entries is compromised, and searches take ℓ steps, then the probability that a given search will not be trapped is $(1-2p)^{\ell}$, and the probability that all of s independent searches will be trapped is $(1-(1-2p)^{\ell})^{s}$. If we assume that $2p\ell < 1/2$, then $(1-(1-2p)^{\ell})^{s} < (2p\ell)^{s} < 2^{-s}$, and the last quantity is no more than p if $s \geq \log_2(1/p)$.

One can easily formalize the above into a proof of the following:

Theorem 1 *Let p be the compromised fraction of the address space, and assume that $p\ell \leq 1/4$, where ℓ is the maximum number of steps in a search. Assume also that each element in the routing tables of every good ID is obtained by choosing with probability $2p$ an arbitrary bad ID, and with probability $1-2p$ the good ID closest to a point chosen uniformly and independently at random from the appropriate, uncompromised address space subarea. Consider, for $s \geq \log_2(1/p)$, the closest ID returned by s searches, each initiated by a distinct*

ID, for the same address x chosen uniformly and independently at random from any given subarea. Then with probability at least $1 - 2p$ this ID is a good ID.

Thus, we are essentially looking at a small redundancy of $\lg(\ell)$, where $\ell < \lg(n)$ is the number of steps in a search, in each step to populate the routing tables; this is enough to guarantee that routing table entries do not get poisoned. Note that this also requires care, however, when adding entries by "casual contact" if this means ejecting previous entries. New entries should be added only by an explicit search, or if they would fit in an underpopulated subarea of the routing table.

5.5 Summary of changes

Summarizing, these are the changes that are required to secure a DHT through the use of the *getSeed* and *isOldWorker* primitives.

1. Resources must be regularly repositioned every epoch to a pseudorandom point in the address space produced by a hash of the shared value obtained by *getSeed*; possibly with $m-$fold redundancy.

2. An ID should be considered active by any node it interacts with and allowed to store values or enter routing tables only after proving it *isOldWorker*.

3. Routing tables should be populated proactively by repeating searches for the same random point in the address space $s \approx \log \ell$ times, where $\ell = O(\lg(n))$ is the maximal number of steps in searches, and taking the closest node returned by the s searches to the target address.

4. Routing tables should be populated reactively, by inserting nodes contacted through means other than the above, only if it does not entail ejecting current table entries.

5. This tolerates a fraction of malicious IDs up to $O\left(\frac{1}{\ell \log(1/\ell)}\right)$ (by Lemma 1), ensuring that a resource is obscured with probability at most $2^{-m/2}$.

New DHT primitives through a Block Chain 6

This chapter will provide a basic knowledge about what block chains are and how a block chain system can be adapted, providing a back-end for the new DHT primitives described in Chapter 4. The first section we provide a description of the general mechanics of block chain. Then, in the second section, we explain how a block chain may be used supporting the new DHT primitives. Finally in third section we give a short history of block chains development as well as depiction of some block chain actually deployed.

6.1 A generic block chain model

The very first block chain was Bitcoin [67], that will be described in following Section 6.3.1. In this section will be described a generic block chain system, how it works and what are the mechanics that make data stored in such a system publicly available while preventing their alteration.

Ultimately a block chain is a distributed system or, from a different point of view, a distributed database. This database allows to store and verify electronic *transactions* (regardless of what exactly a *transaction* represents) without requiring trust among the peers contributing to the system.

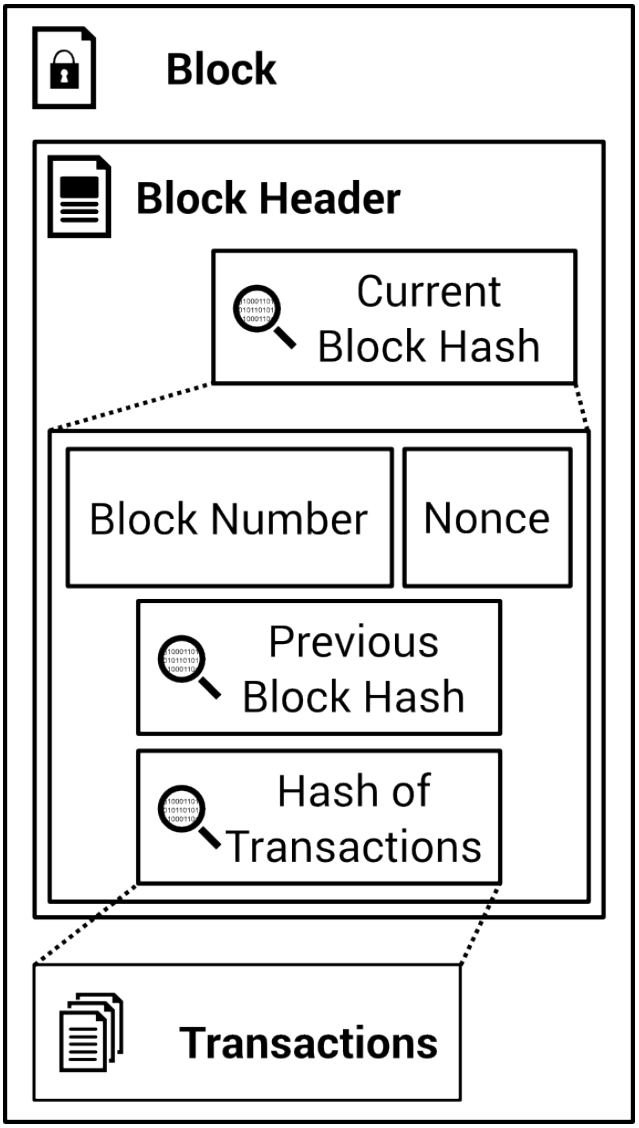


Figure 6.1: A graphical representation of a single block chain *block* and all its basic components.

The basic elements of a block chain are its records, called *blocks*. The block chain is, as its name suggests, a continuously growing collection of such *blocks*. Each *block* record is a more complex data structure, made up of an *header* and a collection of *transactions*. What makes a block chain a secure system is the requirement of some kind of proof of work, that is embedded in its basic elements.

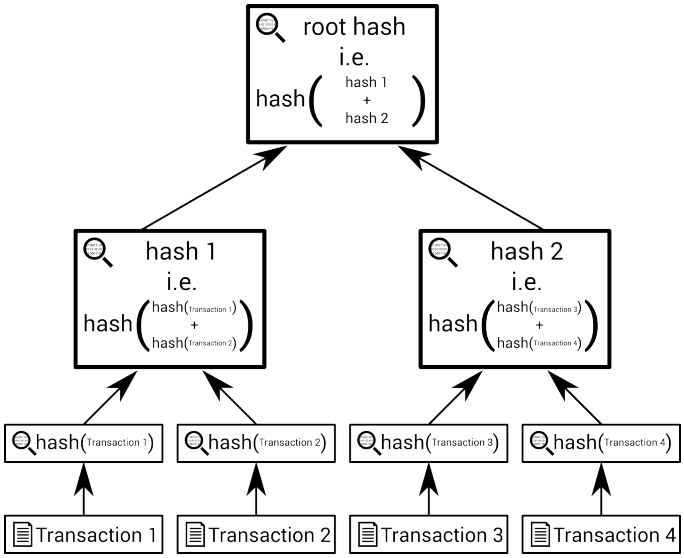


FIGURE 6.2: An example of a binary Merkle Tree built on *transactions*. The root node is a convenient digest, representing the whole *transactions* collection's *hash* in the *block header*.

Figure 6.1 depicts a block chain's *block* structure. In the following paragraph we will detail the basic data structures making up a *block* and how they are used.

The collection of *transactions* may be organized in any fashion as long as it sports the possibility to digest it in a compact form, an *hash* value. For the purpose of simplified *transaction* verification (discussed later in this section), usually *transactions*' collections are organized in the form of a binary Merkle Tree [64], with the tree's root serving as the collection's digested *hash*. Figure 6.2 shows a typical block chain's transaction Mekle Tree.

Each *block header* must be a structure that can be digested as an *hash* as well. The *block header* identify its *block* and should contain at least the following elements: the *block*'s incremental number, representing the *block* position in the overall block chain sequence of *blocks*; the *hash* value representing the set of *transactions* collected in the *block*; the *hash* value of the previous *block*'s *header* – i.e. the *hash* of the *block* whose *header*'s number equals the current *block*'s number minus one. All these elements must be used as arguments to compute the *block header*'s *hashed* value. The *hash* value of previous *block*'s *block header* then represents the link between two

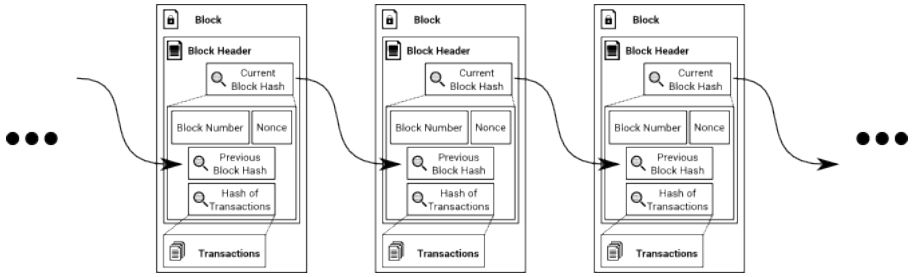


Figure 6.3: An example of three *blocks* from an hypothetical block chain. The link between two subsequent *blocks*, consisting on the predecessor's *header*'s *hash*, is represented by the arrows.

blocks, as shown in Figure 6.3. The term *block header* is often referred to as *header*, as a shorthand.

Please note that the *header*'s *hash* works as well as an identifier for the whole *block*. The *block*'s potion in the overall sequence cannot be altered: modifying any of the components shaping the *block* would result in a different *hash*. Among these elements lies also the previous *block*'s *hash* value. Thus once a *block* is "built" to change its predecessor requires to "re-build" the *block* from scratch. This has a clear implication: in a chain of *blocks*, to be able to modify any of the parameters stored in any *block* requires the modification of all subsequent *blocks* in the chain – i.e. it would require to compute them all anew.

Thus, later modification of data stored in the block chain can be avoided requiring a proof of work embedded in the *block*'s generation itself. This makes the effort required to change a datum already stored in a *block* equal to the effort required to generate a set valid proofs of work: for the targeted *block* as well as for all subsequent *blocks*. Consequently any new *block* added to the block chain contributes to ensuring an ever-growing security to all previous stored *blocks*, increasing the cost required to modify older *blocks*.

Block chain systems usually enforce the proof of work scheme via *block header*'s *hash* computation. The most commonly adopted mechanism it obtained imposing some constrains on the acceptable – i.e. valid – *hash* values. A common proof of work used in block chains is based on the HashCash [9] mechanism. An additional element, called *nonce*, is usually incorporated into a *block header*;

the "correct" *nonce* is necessary to create a new valid *block*. Each possible *Nonce* must be tried sequentially producing the various possible *block headers* until, eventually, an *hash* turns out to be valid according to the proof of work constrains. Rules on validity may incorporate a concept of *difficulty*, a parameter calibrating the "target" for a valid *hash*. *Difficulty* may be adjusted according to the current network *hashrate* (the network's cumulative capacity of calculating *hashes*) making the valid *hash* target easier or harder as a consequence of the global performance by network's nodes.

In a block chain system, peers can perform two different tasks. Nodes partaking in the network may be divided in "regular" nodes and "miner" nodes. Note that a single peer node may be working both as a regular and a miner node or, alternatively, just performing one of the two tasks.

A block chain is organized as an unstructured network. Nodes choose connections between each others randomly, generating a network overlay arranged in the form of a random graph. Messages are propagated in the network via a flooding mechanism, broadcasted from each node to its neighbors.

Miner nodes are responsible for the creation of new *blocks*. They constantly gather *transactions* on behalf of the whole network; check (if needed) if available *transactions* are valid and should be incorporated in the main chain; arrange them in a new *block*'s collection; and, finally, use all these information to search the *nonce* that makes the new *block* a valid one – i.e. its *block header* when *hashed* yields a valid value.

Regular nodes' functions are mainly two: the first is to maintain a distributed copy of the block chain, the latter is to grant access and fruition of data stored in it. Frequently block chains implementations further divide regular nodes in *thin* and *thick* nodes. A *thick* node hosts a complete copy of the block chain while a *thin* does not. Whenever necessary a *thin* node retrieves the needed portion of chain from a *thick* one, effectively being a lean client to the block chain system. *Thick* nodes periodically send each other updates on the last known *block* available in their chain copy; if necessary – e.g. a discrepancy between the chain copies of two nodes – updates are shared to synchronize the copies.

As already mentioned before, a lightweight procedure to check if a *transaction* is contained in a certain *block* may be put in place. If the set of *transactions* stored in a *block* is arranged in the form

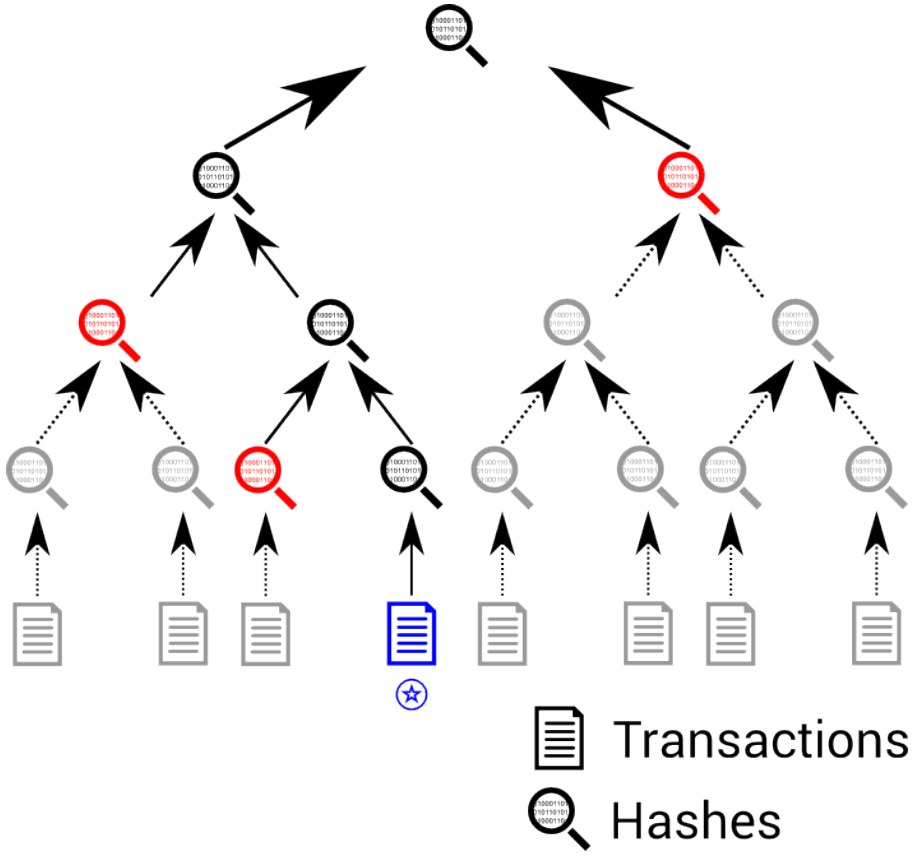


Figure 6.4: A simplified transaction verification is shown in this picture. It is possible to check if a *transaction* is contained in a collection without knowing all other *transactions* if the collection itself is organized as a binary Merkle Tree, just by checking the tree's root *hash*. In this example to check if a certain *transaction* (marked with the star and blue colored) is contained in a collection whose Merkle Tree *hash* is known just three more *hash* values are required to compute the root value: they are shown as the red *hashes*. In the image the whole Merkle Tree is completed (with light gray elements and dotted arrows) for comparison.

of a Merkle Tree, a *thin* node could verify if a given *transaction* is contained in a certain *block* without necessarily need all the data contained in that block. The smallest amount of data needed is just the *block header* and a subset of the actual Merkle Tree: the *transaction* itself; the *transaction* paired with it or, alternatively, its

hash; and the intermediate Merkle Tree nodes' *hashes* all the way up to the root. These data allows for an accurate reconstruction of the *hashed* value of the *transactions*' collection. If the *hash* value of the *transactions* is not coherent with the one available in the *block header*, the latter can not be consistent with its own *hash* value. This is called a "simplified *transaction* verification" and Figure 6.4 shows how it is processed.

Because the system rely on an unstructured network, different *miners* may sometimes simultaneously "discover" different new *blocks* chaining after the same block. If such an event should occur, any *thick* node attach to its own copy of the chain the first valid *block* received. Thus different nodes may experience different endings for the same chain, virtually causing a *fork* in the block chain. *Forks* are settled whenever a longer version of the chain is retrieved: length of a block chain is ultimately a measure of how much proof of work effort has been invested in it. Thus, the longer the chain the more effort the network cumulatively invested in it. Choosing the longer copy over the shorter ones, whenever a *fork* is encountered, simulate a majority voting system: each "unit of effort" (in terms of proof of work) accounts for one vote and the most voted chain is the one upon which the network should share global consensus.

Corrections to the block chain over existing *forks* are expected to happen from time to time. This implies that occasionally chain branches are pruned and, consequently, all *transactions* stored in pruned *blocks* are roll-backed. However, the more a *block* becomes "deeper" in the chain – i.e. the more other *blocks* are chained as its successors – the more unlikely a rollback of such a *block* its likely to happen: to overwrite and outdo an existent chain branch requires an increasingly growing proof or work effort, increasing every time a new *block* is added to the branch itself. Thus once a *block* is sufficiently deep its pruning probability is virtually reduced to a non-existent one, due to overwrite impracticability.

Nodes, regardless of their specific type, may join and leave the network at will. *Miner* and *thin* nodes do rely on *thick* nodes for up-to-date information about the block chain, thus no restriction or special operation is required either at join or at leave time for such nodes. On the contrary, *thick* nodes are the actual hosts of the block chain. This means that when a *thick* node joins the system it should check if its block chain copy needs update or is already synchronized with the shared one. To minimize effort when leaving

the system and then re-joining it, a node returning online accepts the longer (valid) chain as the correct one, updating or overwriting the one already stored locally.

Concluding, a block chain can be considered both a timestamp network and a majority decision system. It works as a timestamp server [31] because it proves that the data must have been already existent (at least) at the time new data was entered. It is a majority decision system because the "correct" chain is always considered to be the longest available valid chain – i.e. the chain that has greatest cumulative proof of work effort invested in it.

6.2 A Block Chain enabling the new primitives

Chapter 4 described a set of primitives that allows the creation of a secure *Distributed Hash Table* (DHT). The main challenge to implement such primitives lies in providing to the whole network either a trusted entity to be queried or a distributed consensus protocol. This section describes how a block chain systems could be tailored to enable a DHT system with *getSeed* and *isOldWorker* primitives. In the following paragraphs we detail a block-chain-based implementation of the two primitives.

The cornerstone of security handed over through the use of the primitives described in Chapter 4 is the availability for every node in the DHT network of a consistent and shared random seed, accessed via *getSeed* primitive. As explained in previous Section 6.1 in a block chain each *block* has (in its *header*) an *hash* value: such value, because of block chain's proof of work requirements, must satisfy some constrains that makes it hard to discover, requiring the sequential tuning of the *nonce* value. This makes it not predictable until the actual *block* is found. Hence a block chain's *header*'s *hash* represents a good candidate for a DHT's seed random value. A good option to attain a quasi-distributed consensus on a random value is the following: let any DHT node able to retrieve (through either a *thin* or *thick* block chain node) some selected *block headers*' *hash* values from the block chain. Some more constrains are necessary, but will be discussed after presenting an *isOldWorker* primitive block-chain-based implementation.

Recall that, as said in Chapter 4, for a node to be "old" and active – e.g. issuing *isOldWorker* on the node DHT *identifier* returns true – requires that the node choose, investing some effort in this process, its *identifier* before the current seed returned by *getSeed* was generated. Moreover tis choice's proof must not be too far in the past. This can be proven rather easily once a block chain is available. We require each DHT node to build a block chain *transaction*. Building this *transaction* will require some effort, and the *transaction* itself must contain the DHT *identifier* and the token representing the work invested. Then this *transaction* must be stored it in a block chain's *block*. The DHT node can choose to mine the *block* itself or rely on other *miners* to build it on its behalf. Now, the criterion to establish is a node is actually "old" and active or not simply reduces to check if it stored a valid *transaction* with its *identifier* in a *block* deeper – i.e. older – than the one currently providing the random seed but, at the same time, not deeper than a certain parameter. Of course to prevent stealing of *identities* "already old", *transactions* should also contain the public part of a *public/private key pair* as additional information. Thus a node will be trusted as old if and only if its *identity* is found in a *transaction* stored in a *block* in the correct "range of deepness" – i.e. coming before the current seed's one but after seed's block minus ε blocks– and, concurrently, it is able to sign its messages using the *private key* verifiable with the *public key* contained in said *transaction*.

As said, *transactions* must require some effort in their building, and a proof of this effort must be included in them. Mimicking the block chain proof of work mechanism, we require a node to provide an integer value included in its own *transaction*. This value must have the property that when hashed with the node's DHT *identifier* the value must comply with some constrains. This mechanism is modeled after Hashcash [9]. Thus generating multiple *identities* would require to build multiple *transactions*, a task mildly computational-intensive.

Clearly, the *transactions* are eventually going to expire – i.e. a *transactions* sink too deep into the block chain sequence. This requires each active node to periodically build a new *transaction*, submit it to the block chain, and start to sign itself using the newly stored *transaction*. Thus any active node is also proofing that its *DHT* activity is being constantly worked out.

As consequence of *forks* happening in block chain accretion,

latest *blocks* may be pruned (see Section 6.1 for details). Because of this in *getSeed* implementation extra care should be taken when selecting the *block* whose *hash* value represents the current seed: a "deep enough" *block* should be used. This will avoid a possible pruning-induced seed exchange and, at the same time, prevent roll back of *transitions* necessary to *isOldWorker* primitive correctness.

Of course requiring a *block* to be "deep enough" implies its *hash* value will be known beforehand when it will become the current random seed. The reader could argue that a such knowledge would make the random seed, in fact, non-random. Albeit a reasonable objection, this is not an issue for the system security. The seed value should be non-predictable just for *isOldWorker* purposes: once the *transaction* is stored in the block chain a node can gain no advantage from the knowledge of the seed.

Thus far we described a method to implement *getSeed* and *isOldWorker*. As explained in Chapter 4 (ans supported by evidence provided in in Chapter 5) a DHT supporting these primitives, and thus a DHT paired with a block chain supporting these two primitives, is protected from a *Node insertion* attack (described in Section 3.2) aiming to hide some specific resources form the rest of the network.

Constantly updating node's *transaction* required by *isOldWorker* provides instead security against a malicious node spawning too many *Sybil* identities (for details on the *Sybil* attack see Section 3.1). A node trying to upkeep a number of *identifiers* is required more and more effort as the number of identities grows (see Chapter 4 and Chapter 5). In other words, a DHT relying on a block chain as the one we just described also provides a mechanism that seriously limit the feasible number of *Sybils* that a single malicious node is able to inject and sustain within the system.

Summarizing, a generic DHT can be enhanced (accordingly with primitives introduced in Chapter 4) by pairing it with a specialized block chain system that will provide a back-end for *getSeed* and *isOldWorker* providing DHT security through randomness, age and work.

6.3 Current block chains

The generic block chain model described in Section 6.1 is based on current implementations of block chain systems. In this section we describe actual block chain systems. We begin describing BitCoin, the first ever realized block chain cryptocurrency, then we go through other prominent examples of existing cryptocurrencies. In the last subsection instead we describe NameCoin, a BitCoin variant that offers a *Domain Name System* (DNS) service, along the traditional cryptocurrency functionality.

6.3.1 BitCoin, the first block chain

BitCoin is the first released and the more famous cryptocurrency and block-chain based system. It was presented in 2008 by a white-paper of an anonymous author writing under the pseudonymous of Satoshi Nakamoto [67]. This subsection will describe this seminal system, that heavily influenced designs that followed it and ignited the cryptocurrency community. We provide a brief history of BitCoin inception, followed by technical details about mechanics peculiar to BitCoin system.

Contrary to popular belief, BitCoin was not the first digital currency nor the first cryptography-based payment system. Indeed in the early 80s blind signature technology was designed, ensuring complete privacy to users conducting online transactions [15]. The author originally designing blind signature was concerned with the public nature and open access to online payments and personal information, and proposed to construct a system of cryptographic protocols preventing a bank or the government to be able to trace personal payments conducted online. This technology was fully implemented in 1990 by DigiCash, a corporation that offered an electronic money service. DigiCash declared bankruptcy in 1998, being unable to successfully cope company growth with user base expansion.

BitCoin, instead, is the first fully decentralized digital currency as well as the first system relying on a block chain. For details on the general mechanisms governing a block chain please refer to Section 6.1. The exchangeable financial value associated to the BitCoin system is referenced to as “bitcoin” or, simply, “coins”.

BitCoin was intended as a payment system as well as a cryptocurrency. Both features are implemented as a *transactions* exchange between BitCoin addresses. Addresses can be generated at no cost by any user of Bitcoin and are alphanumeric case-sensitive identifiers of 26 to 35 characters, starting with the number 1 or 3. To prevent visual ambiguity he uppercase letter "O", the uppercase letter "I", the lowercase letter "l", and the number "0" are never used. Albeit there is no impediment to re-usage of BitCoin addresses, due to privacy and security issues they are not intended to be used more than once.

A *transaction* is a transfer of bitcoin value that is broadcasted to the network and collected into *blocks*. BitCoin *transactions* have input and output values, these values are used to link *transactions* over time. A *transaction* typically references, as its inputs, previous *transaction*'s outputs. It also dedicates all cumulative Bitcoin value coming from its inputs to its new output(s). *Transactions* are not encrypted, so it is possible to browse the block chain and view every *transaction* ever collected into any known *block*. Each *transaction* can optionally include a "mining fee", rewarded to the miner incorporating that same *transaction* in a new *block*. A node submitting a new *transaction* to the system may want to include a fee to expedite its block chain storage: *miners* can choose which *transactions* process first and are likely to prioritize those that pay higher fees.

Ownership of bitcoins means that a user can spend bitcoins tied to a specific address, thus assigning a controlled *transaction* output as input to a new *transaction*. To do so, a payer must digitally sign the *transaction* using a private key specific to the same *transaction*. Without the private key, the *transaction* cannot be signed and bitcoins cannot be spent. This is not reversible: losing the private key credentials makes the *transaction* signed by it (and thus its coin value) permanently lost and unspendable.

Reference previous outputs as new inputs to *transactions* must be verified. BitCoin uses a custom Forth-like[1] stack based scripting system. In each *transaction* the first part of the script – called scriptSig – is paired with each input, while the second half – called

[1]Forth is an imperative stack-based programming language. First developed in 1970, since its virtual machine is simple to implement and has no standard reference, there are numerous implementations of the language.

scriptPubKey – is paired with outputs. The input's scriptSig and the referenced output's scriptPubKey are evaluated (in that order) with scriptPubKey using values left on the stack by scriptSig. The input is authorized if scriptPubKey returns true. Using this system the *transaction*'s sender – i.e. the payer – can create very complex conditions that must be meet to claim the output's value. For example, it's possible to create an output that can be claimed by anyone without any authorization or, on the contrary, it is also possible to require that an input must be signed by ten different keys, or be redeemable with a user-defined password instead of a *public/private key* couple.

A BitCoin wallet stores the information necessary to transact bitcoins. In the BitCoin system the monetary value resides in *transactions* redeemability, implying that bitcoins cannot be parted from the block chain itself. For this reason a better way to describe a wallet is: something that "stores the digital credentials for bitcoin holdings" and allows to access and spend them. BitCoin is a system that features characteristics of as a *Public Key Infrastructure* (PKI): uses a *public key* cryptography. Two cryptographic keys, one public and one private, must be generated by each node. At its most basic, a wallet is a collection of the keys used by the wallet's owner.

Being BitCoin a payment system, double-spending – i.e. the result of successfully spending the same coins in more than one *transaction* – is a crucial issue. BitCoin protects against double-spending verifying each *transaction* added to the block chain, ensuring that the inputs for the a new *transaction* has not been previously spent. While other electronic systems prevent double-spending using an authoritative source approving each transaction, BitCoin uses a decentralized system, relying on a consensus protocol among nodes working as a surrogate for a *Central Authority* (CA). There is some exposure, due to block chain *forks*, to fraudulent double-spending when a *transaction* is first made. The risk decreases as the *block* that stores this *transaction* goes deeper in the chain.

BitCoin's *hash* function is based on the SHA-256 algorithm [23]. For proof-of-work purposes BitCoin uses double SHA-256: candidate data is firstly digested by means of the algorithm, and this *hash* is then *hashed* again through SHA-256 thus obtaining the final result. BitCoin expects that in a certain amount of time the network will be able to create a certain number of new *blocks*: more precisely the system is designed on the expectancy that a new *block* is created

every 10 minutes. This is achieved by tuning the next *block* valid *hash* target through the proof of work's *difficulty*. Valid *hashes* must have leading leading bytes set to zero: the more consecutive zeroes are required, the harder the *hash* is to find. Moreover, every 2016 *blocks* are added in the block chain the *difficulty* will be adjusted to keep the average production of *blocks* at the rate of one each 10 minutes. Every node is able (by chain inspection) to independently infer the correct *difficulty* needed in the next *block* generation.

Miners are encouraged to perform their *block* generation task through an incentive – on top of the transaction fee. Each *block* will contain a special *transaction*: this *transaction* has no input but, on the other hand, has a fixed coin output. Thus a node *mining* a new *block* will be allowed to spend in a future *transaction* an amount of coins obtained as a reward for a previous *block* discovery. Exact coin value of this reward halves every 210,000 *blocks*, approximately every four years. Nakamoto set a monetary policy when he first started BitCoin: there would only ever be 21 million bitcoins in total. This means that eventually the reward will decrease to zero; the limit of 21 million bitcoins is expected to be reached around the year 2140. By that time *miners* will be rewarded solely by *transaction* fees.

The first wallet program – the so called "Satoshi client" – was released as open-source code in 2009 by Satoshi Nakamoto. This is known as the "reference client" because it define the BitCoin protocol and acts as a standard for other implementations. After the release of version 0.9, the reference software was renamed "Bitcoin Core" to distinguish itself from the BitCoin network. Other forks of Bitcoin Core exist.

BitCoin is a digital asset designed to work as a currency. It is commonly referred to as digital currency, electronic currency or cryptocurrency. In February 2015, the number of merchants accepting BitCoin for products and services passed 100,000 [20]. In a nutshell, BitCoin is the first block chain ever devised, and works as a reference for almost all block chain created thereafter. It is a complex system and is used both as a currency and as a payment system.

6.3.2 Other notable block chains

Many other block chains do exist beyond BitCoin. This section provides reviews some of the most relevant systems that stemmed after the original idea of Nakamoto in 2008, alongside a brief history providing context to the birth of such alternative implementations.

Block chain based systems are often called altchains, shorthand for "alternative block chains". Their concept or code (or both of them) are generally based on BitCoin technology. Block chains are used for a variety of purposes, but mainly as a currency just like BitCoin.

After BitCoin inception in year 2009, in 2011 was created Namecoin. This chain is described in following Section 6.3.3. In october 2011 LiteCoin was released and, from there on many other altchains was released. As of December 2016 almost 710 different cryptocurrency do exist, with a total market capitalization estimated around 14 billion dollars. However solely BitCoin assets account for 11 billions of such capitalization and less than 25 altchains do capitalize more than 10 million dollars each.

Litecoin was released on October 7, 2011 by Charles Lee, a former Google employee, via an open-source client available on GitHub. It was a fork of the BitCoin Satoshi client. The main differences between the two systems are on the block chain and currency parameters other than, more importantly, on the proof of work algorithm adopted. Litecoin aims to process one new *block* every two minutes and a half, contrary to the ten minutes period of BitCoin. Litecoins developers claim that this allows for faster "confirmation" of *transactions* – i.e. the *block* hosting the *transaction* is deep enough in the block chain, making its pruning unlikely. However this leads to a higher probability for a new *block* to be pruned shortly after its chain inception. Litecoin network will eventually produce 84 million Litecoins, four times as many currency units as will be issued by the Bitcoin network. Finally, in its proof of work scheme Litecoin uses the scrypt algorithm, a sequential memory-hard function [72] requiring asymptotically more memory than a non-memory-hard algorithm.

Most of the altchains use a proof of work scheme to timestamp their *blocks*. Algorithms used are predominantly SHA-256 [23] and scrypt [72]. The latter currently dominates over the world of cryptocurrencies, with more than 400 altchain implementations.

Other algorithms used for proof of work in other alt chains include SHA-3 [26], Blake [7], CryptoNight [81] and X11 [25].

PeerCoin [44], released in 2012, is the first cryptocurrency that shifted from a pure proof-of-work system to an hybrid proof-of-work/proof-of-stake one. In PeerCoin new *blocks* can be *mined*, through a SHA-256 *hash* proof of work challenge exactly as in BitCoin or, alternatively, they can be *minted*. *Minting* is the process of creating a new *block* through a proof of stake – i.e. proving coin ownership – rather than a proof of work challenge. In PeerCoin's system each *transaction* is paired with the concept of "coin age", representing the time elapsed between the generation of such a *transaction* and its usage as input for a new *transaction*. Note that this is conceptually identical to the time a particular unit of coin has been in possession of the same owner. When a *transaction* is used, its coin age is destroyed and, clearly, a node holding control over multiple *transactions* do have a stockpile of "coin age units". To produce a *block* via *minting* a node must include in such a *block* a special *transaction* called *coinstake*. In the *coinstake transaction* the owner pays himself using some of his *transactions*, thereby consuming associated coin age. Coin age consumption grants the *minter* with the privilege of generating a *block* for the network. The important difference with *mining* is that the *minting hashing* operation is required to search a valid *hash* over a limited space instead of an unlimited search space as in a "normal" proof of work, thus no significantly easing the process. Moreover the system eases difficulty for *minting* process when the *mining* difficulty grows harder: this is done to counterbalance the two strategies each other.

After PeerCoin launched the use of a proof-of-stake-based chain scheme, other cryptocurrencies followed. Prominent examples are ShadowCash [80], BlackCoin [90], NuShares/NuBits[2], Qora[3] and Nav Coin[4]. As a final note, Nxt [66] was the first cryptocurrency to rely its peer consensus purely on a proof of stake scheme.

Finally, after a beginning with a strong focus on financial applications, block chain technology is extending to activities including decentralized applications and collaborative organizations. For

[2] `https://www.nubits.com/`
[3] `http://qora.tech/`
[4] `http://navcoin.org/`

example Steemit[5] combines a blogging site, a social networking website, and a cryptocurrency (known as Steem [48]). Synereo's 2.0 Tech Stack is a blockchain-based decentralized platform, developed by Synereo LTD[6], which is also developing a social network (named DApp) running on the mentioned platform [45].

6.3.3 NameCoin

This section presents NameCoin [57], a project spawned from BitCoin that, other its cryptocurrency application, provides DNS-like services.

Namecoin was the first cryptocurrency created forking the BitCoin software. In September 2010 a discussion was started in the Bitcointalk[7] forum about a hypothetical system called BitDNS and generalizing BitCoin. Inspired by the BitDNS discussion, on April 2011 Namecoin was introduced by forum user Vinced as a multipurpose and distributed naming system based on BitCoin. Namecoin's flagship use case is to provide a censorship-resistant Domain Name System for the top level internet domain `.bit`.

Implementation-wise NameCoin shares code and parameters with BitCoin. For example all of the following details are common to both systems: a double SHA-256 proof of work scheme; expected elapsed time between *mining* of two *blocks*; *difficulty* adjustments; coins rewarded with each new *block*; halving period of rewards; maximum number of coins released.

The main difference in NameCoin is on *transactions*. A *transaction* may include a record consisting of a key and a value, up to 520 bytes in size. Each key is actually a path, with a *namespace* preceding the name of the record. *Namespaces* are online systems that maps names to values, they have different semantics and purposes: `d` represents the *namespace* of DNS records so that the key `d/example` corresponds to the record for the `example.bit` domain. The other currently implemented *namespace* is `id` to manage public online people identities – e.g. tying a real name to an email address, a birthday date and other personal details.

To register a new name – i.e. a record – within a *transaction*, a fee of 0.01 NameCoins is required. This coin cannot be spendable

[5] `https://steemit.com/`

[6] `https://www.synereo.com/`

[7] `https://bitcointalk.org/`

like other Namecoins while it has a name attached to it. Namecoin enforces an expiration time for names. Originally, the time period for a name to expire was set to 12,000 *blocks*, but by March 2012 the expiration period was increased to 36,000 *blocks* – about 250 days. An existing name may be mentioned in an update operation (within another *transaction*) to postpone its expiration, renewing it.

Finaly, other proposed potential uses for NameCoin include a messaging system, personal *namespaces*, notary/timestamp systems, alias systems and issuances of shares or stocks. These functions however are still not available.

Though its purpose as a distributed DNS is interesting, a 2015 study shows that NameCoin reveals itself as a system in disrepair: among the registered Namecoin's domain names analyzed (roughly 120,000), a mere 28 were not "squatted"[8] and have nontrivial content [42].

[8] The term "squatting" referred to a domain name means that it is registered by a user whose utility for that name is close to zero. However who purchased it hopes to profit from selling it to another user whose utility is higher.

A secure DHT implementation 7

We developed a *Distributed Hash Table* (DHT) enhanced with the primitives described in Chapter 4 and backed by a specialized block chain as proposed in Section 6.2 as a proof of concept. This Chapter will describe this system and its implementation. In the first section we present its motivations, while the second section details on its structure and architecture are provided. In the last section of this chapter we eventually illustrate and discuss future improvements of this software.

7.1 An easy-to-use secure DHT

This section illustrates the motives that leaded to the development of *Random-Age-Work DHT* (RAW DHT), an easy-to-use yet secure DHT implementation.

Modern software applications often require some form of a distributed execution environment, be it a cloud or the possibility for two or more application's instances to find and connect with each other. This need has been addressed by a number of commercial middle-layer software – called middleware – that, however, are often cumbersome and require a steep learning curve. Services regarded as middleware usually offer enterprise application integration, data integration, *Message Oriented Middleware* (MOM), *Object Request Brokers* (ORBs), *Enterprise Service Bus* (ESB), *Remote Procedure*

Call (RPC) or *Remote Method Invocation* (RMI).

However available middleware solutions seldom if ever address the following issues: finding nodes; build an overlay network; address nodes in the overlay. Anyway such functions are as important as, for example, ORB marshalling or RPC/RMI tasks. As already discussed in Chapter 2, a DHT comes in handy exactly in this kind of scenario.

Oddly enough though, in spite of the conceptual simplicity of a general DHT (see Section 2.2 for more details) very few library implementations, ready for an out-of-the-box usage, may be found on the Internet. Moreover, those available are vulnerable to the numerous security issues reported in Chapter 3.

Lack of a readily available DHT implementation, possibly secured against the *Node insertion* attack (see Section 3.2), motivated both the theoretical development of the primitives described in Chapter 4 and an actual implementation, RAW DHT, serving as a proof of concept.

More importantly RAW DHT, aside from being a proof of concept, tries to fulfill this need: a code library to ease the development of distributed software relying on a dynamic network overlay.

The following sections will detail implementation choices that lead to a working system, experimentally evaluated in Chapter 8.

7.2 System architecture

This section presents a global view of *Random-Age-Work DHT* (RAW DHT), the system whose development was motivated by arguments expressed in Section 7.1. In Subsection 7.2.1 and Subsection 7.2.2 we respectively describe more in detail the structure of the block chain module and that of the DHT module, while following paragraphs we provide a description of the overall implementation choices that was made during its development.

RAW DHT has been developed in Java: this language is naturally cross-platform and, with small additional coding effort, could be executed on Android tablets and phones besides more traditional PCs. In addition Java language comes with an extensive set of built-in data structures that significantly ease the development of complex systems. To fully take advantage of this specific feature

of the language, the Oracle Java Standard Edition version 8[1] was selected.

Our implementation, made by over twenty thousands code lines, requires as execution environment the Oracle *Java Virtual Machine* (JVM) rather than the OpenJDK[2] one. This is due to the usage of some classes that not included in the open-source runtime. However such classes are used only to render part of a browser-based user interface, and could be easily excluded to make RAW DHT fit for a broader range of systems supporting Java.

Some of the Apache Commons[3] components were used to expedite the code development, choosing structures whose implementation is proven to be solid rather then implementing them anew. Components included in RAW DHT are:

- *CLI* - a command line arguments parser
- *Codec* - a set of general encoding/decoding algorithms – e.g. managing URLs or hexadecimal strings
- *Collections* - extending the Java "Collections" data structure framework
- *Lang* - providing extra functionality for classes contained in `java.lang` package.

Through the usage of the Google Guava[4] library other data structures – e.g. collections – and utilities are accessed, also to speedup the development process as well as diminishing the possibility of new bugs introduction in the code.

A good portion of RAW DHT code was developed following the *Test-Driven Development* (TDD) methodology. TDD requires that any new feature development begins by writing a unit test; then the actual code implementation of the feature is written and eventually it is tested. This practice generally leads to a more robust source code and ensures that tested code is relatively bug-free, drastically

[1] `https://docs.oracle.com/javase/8/`
[2] `http://openjdk.java.net/`
[3] `https://commons.apache.org/`
[4] `https://github.com/google/guava`

simplifying an otherwise painstaking debug process. Testing tools used in RAW DHT development are JUnit[5] and EasyMock[6].

Aside from the two main modules (the block chain and the DHT one, detailed in following subsections) RAW DHT implementation include three more packages worth to be mentioned: logger, DB and settings.

RAW DHT has been designed aiming to provide its functionality on as many platforms as possible. For this reason an *Object-Relational Mapping* (ORM) layer is of paramount importance for the DB module. Thus we implemented for the system a general interface to access its database and it relies on the OrmLite[7] package. RAW DHT natively supports (and comes with) *HyperSQL DataBase* (HSQLDB)[8] – a relational database management system also developed in Java. However by simply implementing a couple of classes, RAW DHT support can be extended to the vast majority of relational databases, from MySQL[9] and PostgreSQL[10] to Oracle[11] or the Android-native SQLite[12].

Logger and settings modules on the other hand provide means to persist data in text files. The latter is used to persists the various modules' parameters. Such parameters may be set up via *Command Line Interface* (CLI) at each software's start-up or, as said, specifying them in the setting files handled by settings module. Logger module, as one would expect, provides a way to prompt system messages both to the software console and to store them in log files. At the moment of writing, such module can produce either plain text or JSON-formatted output files. However it can be easily extended to handle other text formats, extending current classes or implementing required interfaces.

On top of the system described so far we developed a block chain module and a DHT module supported by it. These modules implement an enhanced DHT supporting the primitives described in Chapter 4. In following Section 7.2.1 and Section 7.2.2 these two

[5] `http://junit.org/`
[6] `http://easymock.org/`
[7] `http://ormlite.com/`
[8] `http://hsqldb.org/`
[9] `http://www.mysql.com/`
[10] `https://www.postgresql.org/`
[11] `https://www.oracle.com/`
[12] `https://sqlite.org/`

modules will be described in detail.

7.2.1 The Block Chain module

In this section we describe the block chain module. This module was implemented to provide the RAW DHT module (later discussed in Section 7.2.2) with (part of) the primitives that offer security guarantees to the system (described in Chapeter 4). Block chain module is one possible implementation of the hypothetical block chain described in Section 6.2.

RAW DHT's block chain module is modeled upon BitCoin [67] (see Section 6.3.1 for more details on this cryptocurrency) but, differently from the well-known block chain system, there is no "coin value" associated with it. Moreover the block chain name can be set-up throughout preferences parameters, thus making it is possible to easily create a "private" block chain – separated from the default trunk – just by tuning a parameter via CLI or configuration file.

Block headers are built upon the structure described in Section 6.1. The *header*'s hash is obviously computed on the following elements: the block number; the nonce; the previous block's hash; and the hash of the transactions set. Also the chain's name, a timestamp, the current difficulty and a *miner*'s concur in the hash computation. *Block header*'s hashes are computed as a double SHA-512 [23] – i.e. applying two times the algorithm, the first time digesting the original data and the second time digesting the output of the first iteration thus obtaining the final value.

A *block* object is made up just as an encapsulation of a *block header* object and a data structure collecting the *transactions*. *Block*, *block header* and *transaction* objects are all java-serializable thus a JVM may send them over internet sockets without requiring explicit conversion operations.

Transactions are made up of a DHT *id* (a 64 byte array), a *transaction* nonce (a 64 bit long integer) and the public key of an RSA [35] PKI. This RSA public key is a 2048 bit long key, encoded by the Java Cryptography Architecture according to the format specified in the X.509 standard [36] – more precisely: it is a 294 byte array. Build a *transaction* requires the submitting node to correctly compute the nonce that hashed with a enclosed *id* and the current seed results in a double SHA-512 hash value whose first three bytes

are equal to zero. Please note that the number of bytes set to zero is an arbitrary parameter. A special *transaction* – namely the "null *transaction*" – is created setting all the *transaction* bytes to zero (*id*, nonce and public key byte array). The "null *transaction*" cannot be used by any node to validate their identity. This special *transaction* but is used by miners, as will be later explained.

The collection of *transactions* is organized in a Merkle Tree fashion by a specialized class storing them in an ordered ArrayList object. The same class is used to compute the Merkle Tree root value. Merkle root is then used as the hash value of the *transactions*' collection stored within *block headers*.

Block chain module sports three autonomous services: the *thick* and *thin* node services other than the *miner* node service. At start-up time, accordingly to module's parameters, the node behavior can be selected: *miner* service can be turned on or off and the local node execution behavior can be selected to be as a *thin* or *thick* client.

Miners gather from *thick* nodes (either local or remote) *transactions* that has been submitted by user nodes. Using the *transactions* retrieved a *miner* will try all possible nonce values until a valid *block* is produced. This task can be interrupted whenever a *thick* nodes notify the miner that an equivalent *block* has been found by another *miner* – meaning that currently searched *block* would be rejected when found. If, for any reason, a *miner* node cannot get any valid *transactions* it will still be able to create a new *block* using two "null *transactions*". This is a small implementation trick to prevent the block chain from getting stuck in the unlikely event of the absence of valid *transactions*.

Thin node clients do not store a full copy of the current block chain. Instead, they ask missing data to *thick* nodes whenever needed. When block chain data is retrieved from *thick* nodes it is stored locally, caching it. Note that *thin* nodes do use a database architecture common to that used by *thick* nodes: this way, if a *thin* node chooses to switch to *thick*, download of data already gathered may be spared. A *thin* node provides an interface to access the block chain as well as to submit a new transaction in order of getting it included in a new block.

A *thick* node provides the same block chain access granted by *thin* nodes, but stores a full copy of the chain. *Thick* nodes connect to a number of other *thick* nodes randomly chosen. Also, *thick*

nodes broadcast to known neighbors updates on the block chain. When, due to a *thick* node starting its execution or when a new block is notified to the network, missing blocks are retrieved by neighboring *thick* nodes until the local chain copy is up-to-date. RAW *miners* do notify their existence to *thick* nodes, thus *thick* nodes may act as proxy to the *miners* both for *thin* nodes and other *thick* nodes.

By means of either a *thick* or *thin* node it is possible to:

1. get a block (or its header) from its hash or its block number
2. get the last block (or its header) in the chain
3. search for a *transaction*'s last occurrence
4. check if a specific *transaction* is actually stored in a specific block – identified either by its header, its header's hash or its number.

Block chain module uses both *User Datagram Protocol* (UDP) and *Transmission Control Protocol* (TCP) transport layer protocols. The most appropriate is chosen depending on a trade-off between the amount of data to be sent and the statelessness of the required connection. All socket communications (TCP) do have a timeout set to 30 seconds.

Although completely functional, block chain module has not been optimized. As we will discuss in Section 7.3 a number of improvement may be put in place with a relatively small effort.

Concluding, RAW DHT's block chain module is a proof-of-concept providing a fully operational implementation of a possible back end – according to the one described in Section 6.2 – enabling a DHT with the security guarantees explained in Chapter 4.

7.2.2 The DHT module

This section describes the *Random-Age-Work DHT* (RAW DHT) core module. Relying on the block chain module described in previous Section 7.2.1, it offers a working proof-of-concept of a DHT sporting the security guarantees presented in Chapter 4. RAW DHT has been designed especially to defend against keys' hiding or alteration, as would happen in case of a *Node insertion* attack (see Section 3.2).

RAW DHT is a modified implementation of Kademlia [63]. For more details on Kademlia please refer to Section 2.3.5. Addresses' length has been set to 512 bit and, consistently with this choice, SHA-512 [23] was selected as hashing function.

RAW DHT obviously support all the basic functions sported by Kademlia (*ping*, *store*, *find node*, and *find value*) and, additionally, supports an implementation of *getSeed* and *isOldWorker* – primitives described in Chapter 4.

The epoch seed retrieved calling *getSeed* is, as a matter of fact, just a block header's hash value obtained querying the block chain module. RAW DHT's *getSeed* select as a "seed block" the last block whose number is multiple of a constant B and, additionally, whose depth[13] is at least greater than another constant D. The first condition is needed to identify if a candidate block is, according to its block's number, a "seed block". The second, similarly to BitCoin "confirmations"[14], is needed to reduce the possibility of a "seed subversion" due to a block chain fork overtaking the current chain. In RAW DHT's implementation, at this dissertation is written, has set $B = 7$ and $D = 5$.

Whenever a resource – e.g. a key – needs to be hashed, the seed retrieved with *getSeed* is concatenated to the end of resource's bytes representation. Thus, in each seed epoch the same resource will have different (and unpredictable) hashes.

Each RAW DHT node, when choosing its ID creates an RSA-based *Public Key Infrastructure* (PKI) keys pair. Any DHT node is thus identified by three objects: its ID; a *transaction* nonce; its public key; and its physical IP address. ID, *transaction* nonce and public key must be encoded in a block chain *transaction* (see previous Section 7.2.1). As soon as such *transaction* gets incorporated in the block chain, the DHT node saves the number of the block containing it. Please note that, given a node's ID, the nonce and public key a *transaction* can always be unambiguously reconstructed. If the node's *transaction* is not yet incorporated in any block, the special value "-1" is used until a valid block number could be provided.

[13] In the context of block chains the depth of a block represents the number of blocks following it up to the last in the chain.

[14] In the BitCoin system, to be secure against double spending a transaction should not be considered as "confirmed" until it is at least a certain number of blocks deep.

All RAW DHT messages are fit with the node's transaction block number, all the node's identifiers (ID, *transaction* nonce and public key) and a private-key-signed version of the message itself.

This usage of *transactions* allows three useful behaviors: stored *transactions* can be used to enforce *isOldWorker* policy; they decouple a node from its physical address; they allows to check if a node providing its *transaction* number in a message is its rightful owner.

The *isOldWorker* primitive is thus easily implemented: when a node needs to verify if another node is "older" than current epoch seed and is active – i.e. when communicating with – it will:

1. re-build from received message the other node's *transaction*
2. check if reconstructed *transaction* is actually contained in advertised block – identified by its number, say X
3. if *transaction* is actually contained in advertised block, check if said block's chain position is at an acceptable depth – i.e. for X will hold $L - (B \cdot \eta) < X < L$ if the last known seed block number is L (clearly $L \ mod \ B = 0$ and L must be at least D blocks deep in the chain).

If all of these conditions are satisfied then *isOldWorker* check ends up returning true. η is currently set to the value of 3.

Public and private keys are used by a node to rightfully claim usage of its own ID. This permits nodes to use their "matured" ID independently from the current physical address that may be changed at any time – e.g. by an *Internet Service Provider* (ISP). To do so, each message must contain a private-key-signed version of the carried data. This stops other nodes from pretending to be some, maybe already old, other node.

A node that fails *isOldWorker* test but that can still provide a public key and a *transaction* nonce satisfying the proof of work conditions is not trusted as host for key/value records storage. Anyway it can still ask for storage on other nodes and perform look-ups on keys or nodes. A also failing to provide a valid proof of work is simply ignored and communications with it are immediately dropped.

Being node identities uncoupled from physical address, the boostrap phase becomes a bit more complicated. An initial set of

nodes for a new instance joining the network cannot be easily hard-coded (either in the actual source code or through a user-generated default file): addresses and *transaction* nonces may, obviously, change at any time the 294 bytes public key representation of each node would make it cumbersome and difficult to manage. Instead a "nodes info server" service has been created as a RAW DHT's sub-module, mimicking the servers used to retrieve eMule KAD references – i.e. the well-known "nodes.dat" files[15]. The "nodes info server" are simply DHT nodes listening on a dedicated network socket. As soon as a node info server receives a nodes request, it will just provide a number of references to the nodes in server's routing table. Using this mechanism allows the list of nodes info servers to be as simple as a set of <IP address, port> entries. These short entries con thus be stored on a user-editable default file.

A RAW DHT node's routing table works as proxy a continuous usage of the *isOldWorker* primitive: node references are inserted in it only if they pass the *isOldWorker* check – also signing their messages with the correct private-key signature – accordingly to the routing table insertion policy described in Section 5.4. Thus whenever a node reference is retrieved from the routing table, it is guaranteed that such reference has already passed an *isOldWorker* check. Note that whenever the associated transaction associated to a routing table entry expires, a node fails replying to any message – e.g. a ping – it is presently removed from the routing table.

On node's shutdown all references in the routing table are persisted on a local file. When a DHT node begins execution it populates its routing table through a series of carefully chosen look-ups starting from node references obtained by a nodes info server. Alternatively, if a persisted file is present a node uses DHT contacts saved from a previous software execution. Please note that, in the current software version, whenever an insert operation is done on the routing table an *isOldWorker* check is always triggered.

Look-up on a key's value locates the set of nodes closer – in XOR metric – to the key's hash, remember that the actual hash is modulated by current seed. These values may be also used in the *find value* procedure to retrieve values associated with the looked-up key. Each found node will reply with all values known to it up to a certain limit – currently 300 maximum entries per reply. If values

[15] `http://www.nodes-dat.com/`

known to a replying node exceeds the limit, returned values are randomly chosen among the available ones.

RAW DHT, unlike many Kademlia implementations currently available, communicates also on TCP rather than relying only on UDP. More precisely messages exchanged during a *store* procedure and in the final step of *find value* are sent over a TCP channel. This leverages on the native JVM serialization/de-serialization feature allowing to black-box the operations required to send objects representing (the list of) value(s) involved in such procedures.

As a final remark, it is very important to note that the whole RAW DHT *transactions* cannot be piggybacked on existent block chains. In particular, we cannot insert our required data in a NameCoin's value field – see Section 6.3.3 for a description of this block chain. NameCoin's specification requires that its value fields must be UTF-8 encoded JSON objects with a maximum size of 520 bytes[16]. Instead a RAW DHT *transaction* formatted as a JSON object is about 800 bytes long[17].

Albeit not optimized, current RAW DHT implementation is a completely operational DHT. Moreover, using the block chain described in Section 7.2.1 RAW DHT supports all the new primitives presented in Chapter 4. RAW DHT is a proof-of-concept for the a new DHT model, secure against attacks tampering on selected keys (see Chapter 3 for some attack examples).

7.3 Future work

This chapter described RAW DHT system, a proof-of-concept implementation of the novel DHT proposed in Chapter 4. This last section will discuss some of the most important improvements that could be included in future versions of the software.

A considerable improvement can be attained developing a more accurate management of network messages: currently all incoming messages are processed sequentially, potentially leading to occur-

[16]Both for domain names `https://wiki.namecoin.org/index.php?title=Domain_Name_Specification#Value_field` and for generic identities `https://wiki.namecoin.org/index.php?title=Identity`.

[17]In the current implementation it size may vary between 760 and 778 bytes, depending on the nonce included. Note that this is a tight representation, shorter than the same object formatted according to a JSON pretty print.

rence of unwanted delays. Note that, although TCP connections are more reliable and automatically manage data integrity checks through the Java platform, some RAW DHT messages – e.g. *store* requests and *find value* replies – could be converted to UDP speeding up the whole exchange.

In the DHT module, for simplicity reasons, the code implementing *isOldWorker* is called whenever such a check is necessary. The same code is also used at start-up time, when a node populates its routing table for the first time. Contacts used during routing table initialization can be retrieved from the network or from a locally persisted file at the end of a previous execution. Note that persisted node references has been already checked with *isOldWorker* before current software initialization, during the last software usage. RAW DHT software would surely benefit from a more efficient and easily implementable routing table bootstrap, leveraging on the fact that locally retrieved contacts can be accepted just (locally) checking if associated block number is still valid – i.e. didn't sink too deep in the block chain. Also note that in some cases a node could need to perform *isOldWorker* check on more than one node at the same time: when, at bootstrap time, a list of new nodes is retrieved or when one of the intermediate node *look-up* steps returns a set of new nodes. Currently for each node is independently performed an individual check. When a *thick* block chain node is executed on the local machine *isOldWorker* check is relatively quick – just a database access. However when the local block chain client is a *thin* node the block chain request must be sent out to the *thick* ones. This means that (up to) three randomly chosen *thick* nodes are selected from the local node and a query message is sent to each one of them. This process is then repeated for every DHT node that needs verification. Clearly, allowing to check more than one DHT node with the same message would lead to a significant performance improvement in the authentication phase.

In the block chain module when a *thick* node begins its execution, it asks to other know *thick* nodes the last known block header before starting its normal tasks. If this block header matches the last available one stored in the local database than no special operation is required and the main task loop begins; otherwise a block chain update is needed. The starting node randomly select one of its known neighbors and ask for missing blocks. Currently RAW software, albeit using the same TCP channel to save connection time,

sequentially asks each needed block until the chain is completely updated. Note that this doubles the amount of messages exchanged – one request for each block received – and could be improved asking beforehand all the (supposedly) needed blocks. Additionally if the update fails for any reason, block chain updating task is dropped; then execution is started with the currently available block chain copy. Mind that as soon as a new block will be discovered and flooded to the network the chain copy will be updated. Nonetheless a more sophisticated block chain bootstrap update procedure may be devised and implemented investing a small development effort.

Anyway note that the block chain module is not irreplaceable. Thus above every other possible improvement, a different back-end for the primitives presented in Chapter 4 may yield a radical performances boost. The major benefit coming from a block chain based system is that it represent a distributed authority, trustworthy mainly for two reasons: any node can join its backbone – as a *thick* node; and the block-chain-stored data are hard to counterfeit. Moreover, considering only the *thick* nodes' network, a block chain system is less scalable than a general DHT. RAW DHT is built on a block-chain-based architecture lacking a better solution for the (quasi) distributed consensus problem. It is important to note though that any system providing a network-wise access to an unanimously accepted random string would be an equally viable alternative.

We described some of the improvements that RAW DHT software would get advantage of. Clearly those are just the most important ones as would be tedious to compile a list all the small data structure improvements that could (and will) be made on the current code-base. Nevertheless these improvements demonstrates, to some extent, that albeit RAW DHT is still a proof-of-concept software, it could easily be turned to a fully fledged stable DHT system.

Experimental evaluation 8

In Chapter 7 was presented RAW DHT, an implementation of the novel secure DHT system presented in Chapter 4. To validate its usability we performed a series of experiments, which are described in this chapter. In the first section we describe the test bed used for RAW DHT evaluation – the hardware and its configuration – as well as each of the experiments carried out. Then, in the second section, we present and discuss the outcomes of these experiments.

In the light of these results the proof-of-concept RAW DHT software proves that the enhanced DHT concept of Chapter 4 can be a viable option in a real-world scenario. Moreover RAW DHT itself, with some optimization and improvement, can be a sensible choice in contexts requiring the decentralization of a DHT layer as well as some guarantees on the integrity of stored data.

Note that in the following paragraphs of this dissertation with the term "*thick* node" or "*thin* node" we may refer to two different things. It could properly denote a *thick* or *thin* block chain module's node or, more in general, a full RAW DHT software instance running its block chain module as a *thick* or *thin* client. The context will help recognize which of the two meanings is the correct one.

8.1 Experimental set up

This section presents both the machines used as test bed for RAW DHT and the experiments performed. While the setup and specifics of the experiments are detailed in the following paragraphs, the results of these test runs will be later discussed in Section 8.2.

Eridano cluster

Machine name(s)	No. of machines	CPU	No. of cores	No. of threads	Clock freq.	RAM size
Stargate	1	AMD Sempron 140 (C2)	1	1	2.7 GHz	4 GB (2x DDR3 2048 MB)
Eridano10	1	Intel Core i7-950	4	8	3.06 GHz	24GB (4x DDR4 4096 MB)
Eridano11 to Eridano25	15	Intel Core i7-950	4	8	3.06 GHz	18GB (3x DDR4 4096 MB + 3x DDR4 2048 MB)

Table 8.1: Eridano cluster system specifications.

A variety of machines widely differing both in processor power and memory available were used, attempting to simulate a real-world environment. First and foremost Eridano cluster hosted the majority of the RAW DHT instances that made up the experimental network. Eridano is a computer cluster made up of sixteen computational nodes (namely Eridano10 to Eridano25) and an access master node (Stargate). Table 8.1 shows memory and CPU power of each

of the machines making up the cluster. Stargate though did not host any RAW DHT instance during any of the experiments performed: this node was used as *Network Address Translation* (NAT) box routing communications to and from Eridano nodes.

Additionally seven more personal computers were used. These machines are more varying in terms of specifications, that are detailed in Table 8.2. Six of them joined RAW DHT experimental network, while the last one – namely "Wrong" – was used as control node managing all the instances executed: issuing them start and stop commands as well as requesting DHT operations – e.g. storing keys or looking them up.

The first experiment performed was meant to check that the block chain module worked properly. This run produced a block chain about 400 *blocks* long. Up to half – *block*-wise speaking – of the experiment new *miner* nodes were added to the network, a new one joining after a random number of seconds ranging from 3 to 120. Then, for the other half of the run, the number of *miners* was not changed. During this experiment the number of block for a difficulty adjustment was lowered form the standard 2016 *blocks* – as in BitCoin – to 100.

Other experiments measured start-up times – DHT module's; block chain module's; and overall time from software starting to front end usability – and DHT records store times. Note that the time for for a store procedure is a valid upper bound to look-ups: a record is actually stored on the set of nodes returned by looking up the record's key itself.

The first batch of these experiments was performed making a RAW DHT node join a stable networks of various size (sizes was 2, 4, 8, 16, 32 and 64 nodes). Joining node was equipped either with a *thick* or a *thin* block chain client node. Both the *thick* and *thin* equipped node joined the network 300 times, each time executing RAW software for a random number of minutes ranging from 1 to 5.

In experimental conditions equivalent to those presently described, both a *thick* and a *thin* stored a fixed number of keys on the network during each run. Then, during the same run, the stored set of keys was both looked-up and their related values retrieved. This was done to test for search accuracy and correctness.

The next set of experiments measured times on each node participating in RAW DHT networks whose sizes varied over time –

PCs testing RAW DHT software

Machine name(s)	CPU	No. of cores	No. of threads	Clock freq.	RAM size
Nehalem	Intel Core i7-950	4	8	3.06 GHz	12 GB (6x DDR4 2048MB)
Wrong	AMD Phenom II X4 840	4	4	3.20 GHz	8 GB (4x DDR2 2048MB)
Eolo	Intel Pen-tium4	1	1	3.00 GHz	2 GB (2x DDR 512MB + 1x DDR 1024MB)
Cafe	AMD Athlon II X2 250	2	2	3.00 Ghz	4 GB (2x DDR3 2048MB)
Chievo	Intel Celeron 430	1	1	1.80 GHz	2 GB (2x DDR 1024MB)
Nemesi	AMD Athlon 64 X2 5600+	2	2	2.80 GHz	4 GB (2x DDR3 2048MB)
Psiche	AMD Athlon 64 X2 5600+	2	2	2.80 GHz	4 GB (2x DDR3 2048MB)

Table 8.2: System specifications of personal computers used to test RAW DHT software.

within the same experiment run. The maximum number of nodes making up such networks was again 2, 4, 8, 16, 32 and 64 nodes, but participating nodes randomly disconnected or re-joined the network creating a very high churn: the maximum number of simultaneously failing nodes ranged from 35% up to 87%, these numbers are relative of the maximum number of nodes allowed during each experiment. In this last batch of experiments half was performed running only block chain *thick* nodes, while the other half ran on a network of mixed *thin* and *thick* nodes.

Table 8.3 summarized the individual and cumulative duration of said experiments. Data was gathered from RAW DHT instances for a period of 22 days cumulatively.

Lastly, before turning to result discussion, some values for the string key "Lorem ipsum dolor sit amet" was stored across various seed epochs. Key's ID during each seed epoch was recorded to monitor the resources' random positioning.

Execution times

Experiment	**Time length**
Block chain, blocks production	42 hours 11 minutes
Thick node joining stable networks	166 hours 22 minutes
Thin node joining stable networks	199 hours 58 minutes
Churning networks (only thick nodes)	52 hours 6 minutes
Churning networks (mixed thick and thick nodes)	59 hours 13 minutes
Thick node search accuracy	2 hours 16 minutes
Thin node search accuracy	6 hours 35 minutes
Total	528 hours 42 minutes
Total in days (rounded)	22

Table 8.3: Duration of the experiments performed running RAW DHT software.

8.2 Performance evaluation

With this section we present and discuss results of the experiments described in previous Section 8.1. Although the single experiments was overall described in previous section, to make results more comprehensible some more explanations on the experiments performed will be added throughout the results discussion.

Before diving in the details, we would like to begin addressing the cumulative experimental time. As shown by Table 8.3 RAW DHT software has been extensively ran in laboratory conditions. As a general consideration, a reasonable requirement for a distributed software is to correctly work for long periods of time. The extensively period of time during which our experiments was performed, was meant also to prove RAW DHT's ability to sustain prolonged work.

Data evaluation is divided in two subsections. We analyze results about the block chain module in the first subsection, while in the second we discuss the DHT core module's performance.

8.2.1 Block chain related tests

A 400 long block chain has been created in an experimental environment, steadily increasing the number of miners nodes participating in the network. The miners continued to increase up to the maximum number of 64 nodes until the chained reached the length of 200 blocks. Then the miners network continued, producing the last 200 blocks of this test block chain in a steady regime.

Figure 8.1 shows the amount of time spent finding each of the blocks. In the first half of the experiment – shown on the left of the red dashed vertical line – time required to mine a block is clearly shorter: this is due to the steady increase of overall computation power available to the network. The second half of the experiment shows some noisy measurements, due to the fact that only two difficulty adjustments occurred – between block 200 and 201 and between block 300 and 301. Difficulty corrections are known to be prone to overshooting – as well as undershooting – especially when there are few miners nodes involved in a network[1]. Nonetheless

[1]Remember that mining of a node is, in essence, a process based on pure chance: a block can be either find immediately or after a long time. The more miners involved, the more the average block discovery time tends to regularize and conform to an consistent value.

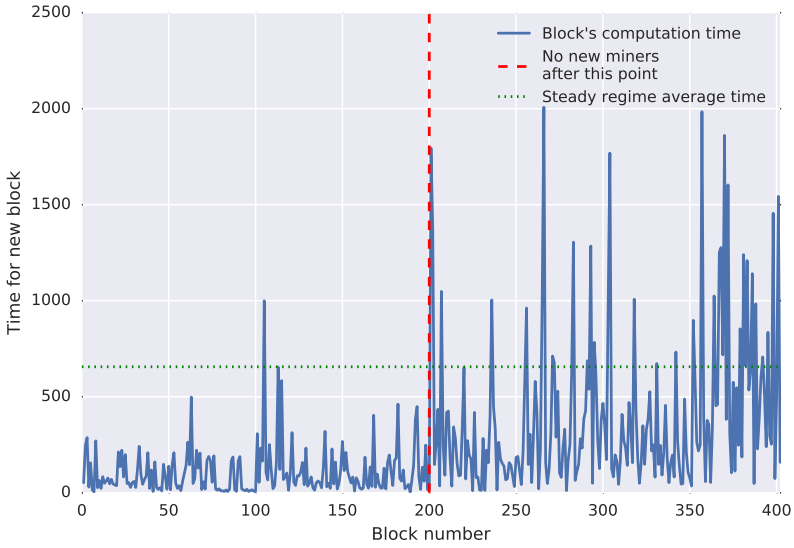


FIGURE 8.1: Blocks' computation times during block chain test. The red vertical dashed line marks the point after which no new miner nodes were spawned in the network, leaving it in a steady computation regime. The green dotted horizontal line shows the average block's computation time during the second half of the experiment (after block 200).

during the second part of the experiment, in spite of the jitter involved, the time spent in the calculation of a single block is on average 655.93 minutes – marked by the dotted horizontal green line. This is coherent with the block chain parameters: time expected to produce a new block was set to 600 seconds.

As described in Section 7.2.2, a new seed is rolled out by the block chain module every seven blocks. This implies that the expected seeds intertime would be 4200 seconds. In fact, considering the latter 200 blocks when the mining network was in a stable condition, the recorded intertime between two seed-blocks averages to 4494.66 seconds.

Using the seeds generated by this example block chain, the key's IDs generated using the string "Lorem ipsum dolor sit amet" was mapped on the full DHT address space. Figure 8.2 maps the relative position of obtained IDs on the ordered address space against the

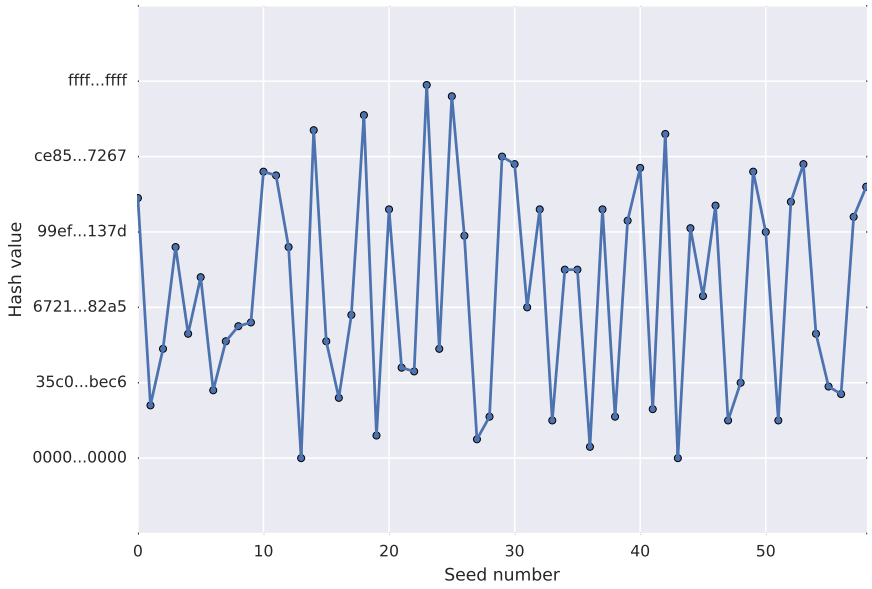


Figure 8.2: Plot of how the hash value of key string "Lorem ipsum dolor sit amet" was changed by random seeds during the performance tests.

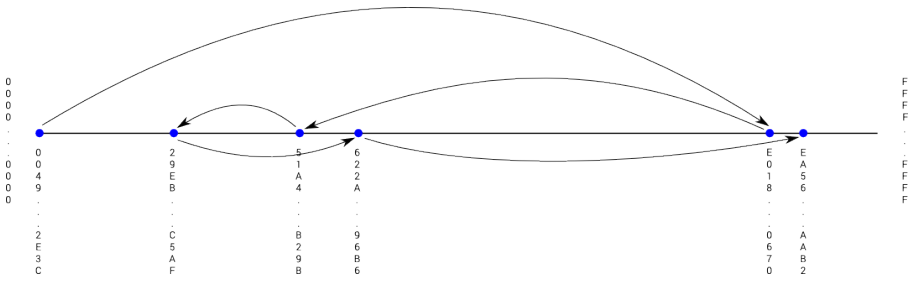


Figure 8.3: Pictorial representation of how the hash value of key string "Lorem ipsum dolor sit amet" was changed by random seeds ranging from seed number 12 to seed number 17 during the performance tests.

seed number – equivalent to a seed epoch. Figure 8.3 is a graphical representation of the relative positions and movement from one to the next one of a subset of such key's IDs – specifically those obtained using seed 12 to seed 17. Moreover we evaluate the distance – as the number of IDs – between all the calculated IDs for string "Lorem ipsum dolor sit amet". We compare this distances with the expected distance of an equal number of uniformly spreaded IDs. Relatively to the theoretical uniform distribution, the average distance in the actual distribution of IDs is short just of a mere 1.689%.

In light of these data the block chain module of RAW DHT appears to be a sufficiently reliable provider of random seeds, as expected by an implementation of the specifically purposed block chain described in Section 6.2.

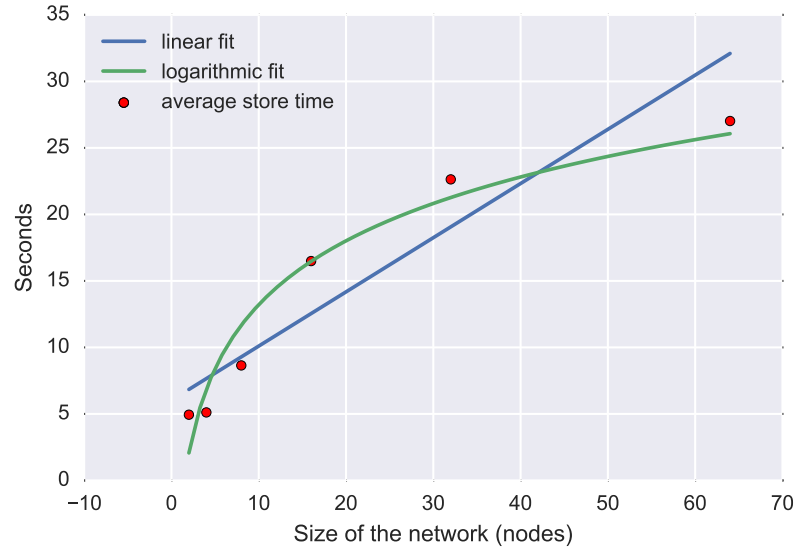
8.2.2 DHT tests

Two sets of experiment were executed making a RAW DHT node (running the block chain module either as a *thick* or a *thin* node) join stable networks of varying sizes. The first set measured startup and store times of a RAW DHT node. The second set instead measured look-up times and their accuracy. Finally measurements was made on churning networks of varying sizes (half made up only of *thick* nodes and the other half made up of mixed *thin* and *thick* nodes) to be compared with the data gathered on stable networks. in this section we present and discuss data from aforementioned experiments.

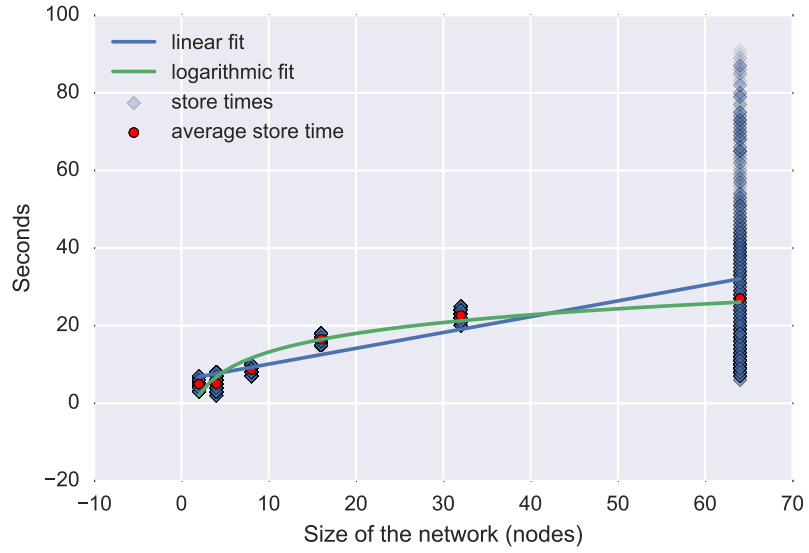
Times were recorded storing keys on networks of varying sizes. These networks was made up of 2, 4, 8, 16, 32 and 64 RAW DHT nodes. Both a node running a *thick* and a *thin* block chain instance performed 300 key stores on the various networks, recording the time spent in the process. An *Ordinary Least Squares* (OLS) linear regression model (calculated with Python's module StatsModels[2]) was used to compute on such data both a linear fit and a logarithmic fit in function of the number of network number of nodes.

Figure 8.4 and Figure 8.5 shows the data and the computed fit functions. However to ease figure reading, the samples are shown only on the (b) sub-figures, while (a) sub-figures shows

[2] `http://statsmodels.sourceforge.net/`

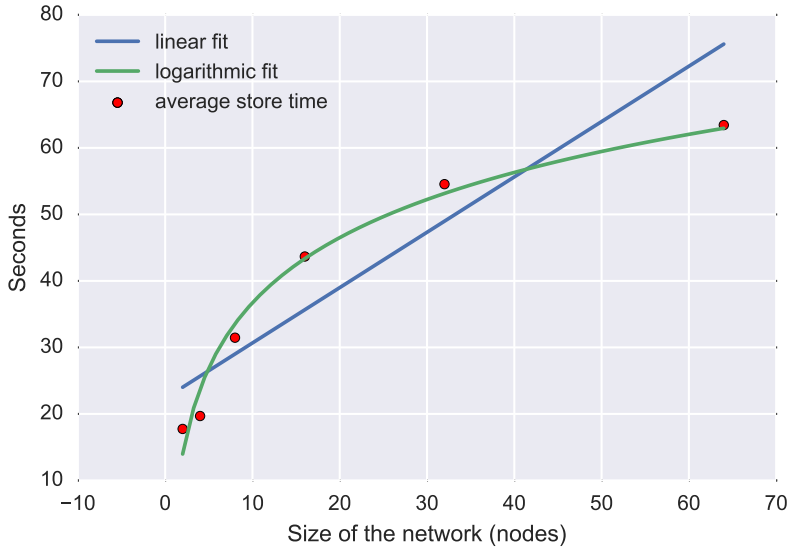


(a) Showing only average values

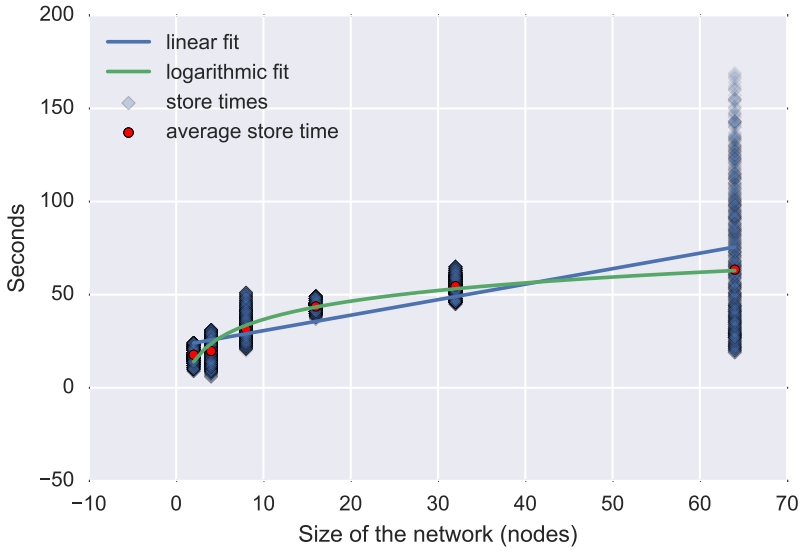


(b) Showing all sampled values

Figure 8.4: Store times recorded on a RAW DHT node running a block chain module *thick* node. Both plots show the linear and logarithmic fit to the data. (a) only shows mean values, while (b) shows also all the sampled values.



(a) Showing only average values



(b) Showing all sampled values

FIGURE 8.5: Store times recorded on a RAW DHT node running a block chain module *thin* node. Both plots show the linear and logarithmic fit to the data. (a) only shows mean values, while (b) shows also all the sampled values.

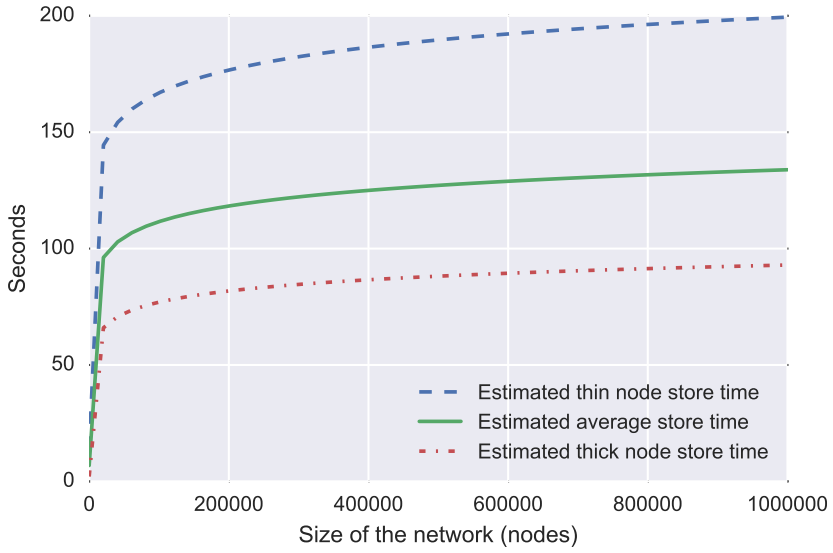


Figure 8.6: Estimation of store times on a RAW DHT network sizing up to 1 million nodes. The dashed and dot-dashed lines are the plots of logarithmic fits shown in Figure 8.4 and Figure 8.5. The solid line instead, is the plot of the OLS logarithmic fit on the whole body of data.

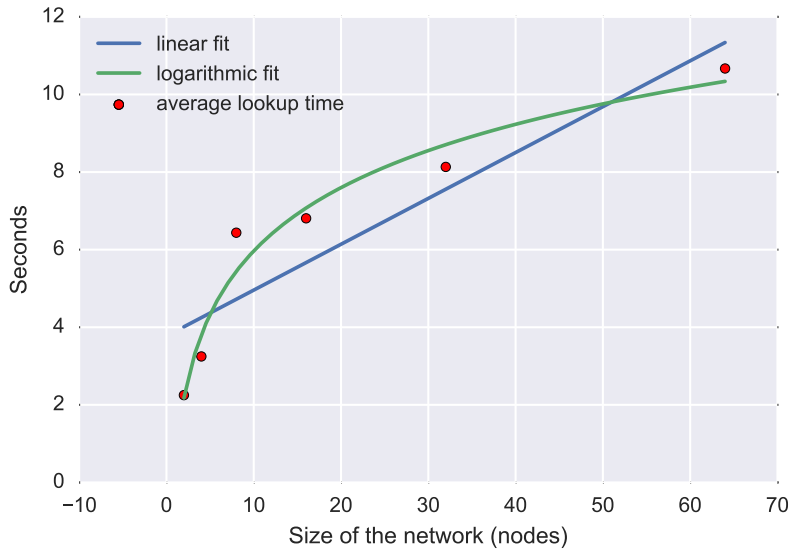
only the mean values. Note that sampled times in the 64 nodes networks do fan out substantially. This is due to routing tables buckets getting more and more full: when retrieving desired storage locations Kademlia's routing algorithm may lead to some "lucky" short (and fast) routes as well as "unlucky" slow routes. On smaller networks instead routes are more consistent and times are generally dominated by mandatory *isOldWorker* checks that, on the other hand, are fewer in number and thus keep the maximum store times relatively low. Also note that measured times include network latencies on top of the time time spent both by routing itself and completing *isOldWorker* primitive operations.

Although the model coefficient of determination R^2 between the logarithmic and linear fit is somewhat similar (differences range in $[0.05, 0.15]$), the number of routing hops – that is also the number of *isOldWorker* checks performed during a query – is expected to be logarithmic in the number of nodes participating to the net-

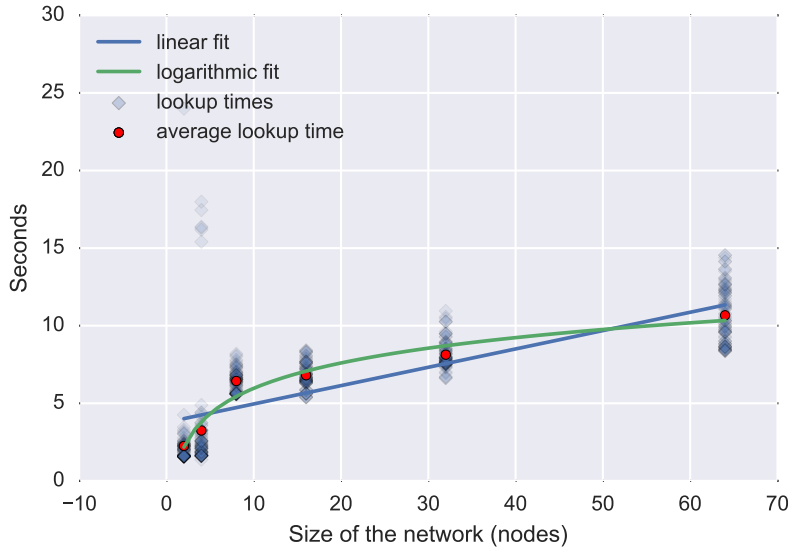
work [63]. Given the impossibility to test RAW DHT software in a real test network whose size would be comparable to a world-wide deployment scenario, we do not have at our disposal real data on how RAW DHT would perform in a broad network. Nonetheless, it is still possible to estimate how RAW DHT would perform using the fitted logarithmic curves shown in Figure 8.4 and Figure 8.5 or,using the whole data (both from *thin* and *thick* nodes), making an OLS logarithmic fit describing an average behavior. In Figure 8.6 we shows the projected estimated store times based of these fits. Albeit estimated time is not low, the following must be considered: the current form of RAW DHT software is absolutely *not* optimized. Moreover the expected store times can still be considered affordable, in exchange for the keys' security attained by the mechanics described in Chapter 4 and Chapter 5.

A Kademlia-like DHT store operation does include a prior look-up. For this reason another set of experiments was performed in the same fashion of aforesaid store experiments. In this set-up, having stored on the network a set of keys, a node running a *thin* block chain client and a node running a *thick* client performed operations looking up the keys. Times spent looking up keys was recorded and fitted with an OLS model as did for the store times. As in the case of store times, we expect the logarithmic fit to be correct due to Kademlia's logarithmic behavior described by its original work [63]. Look-up average times, complete samples plot, linear and logarithmic fits for *thick* and *thin* nodes are shown respectively in Figure 8.7 and Figure 8.8. Again, the fan-out visible in sub-figures (b) is not surprising, as explained discussing the store times data. By comparing data observed for the look-ups and the storage operations – i.e. comparing Figures 8.7 and 8.8 to Figures 8.4 and 8.5 – recorded times reveal that store operation does have an overhead in respect to look-up. For the current RAW DHT software this behavior is expected: to store a key/value record onto the DHT, a node first perform a look-up on the key's ID thus locating the set of recipient nodes. Then the final value storage operation is initiated, thus triggering a final *isOldWorker* check.

Regardless of the recorded times during the look-up experiments, it is worth mentioning the accuracy of look-up and value retrieval. In these experimental setup the probing node performs multiple searches of a set of keys. Having previously store such keys, probing node is beforehand aware of the results that should be obtained

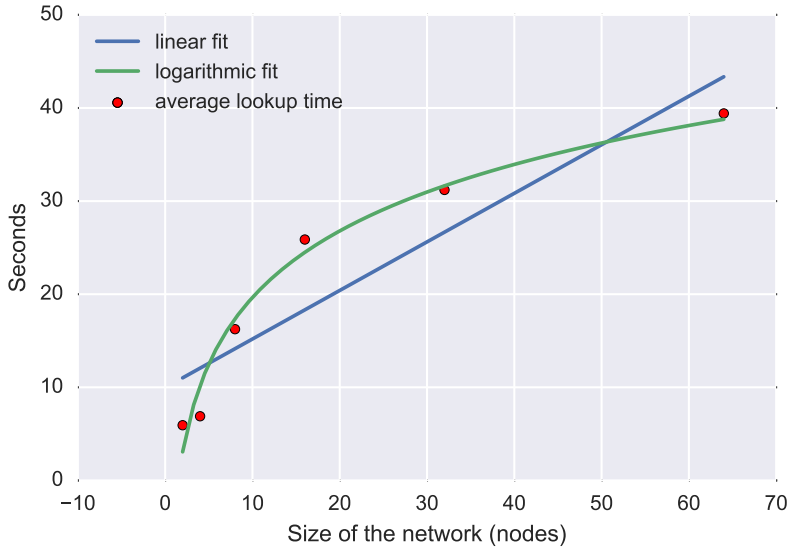


(a) Showing only average values

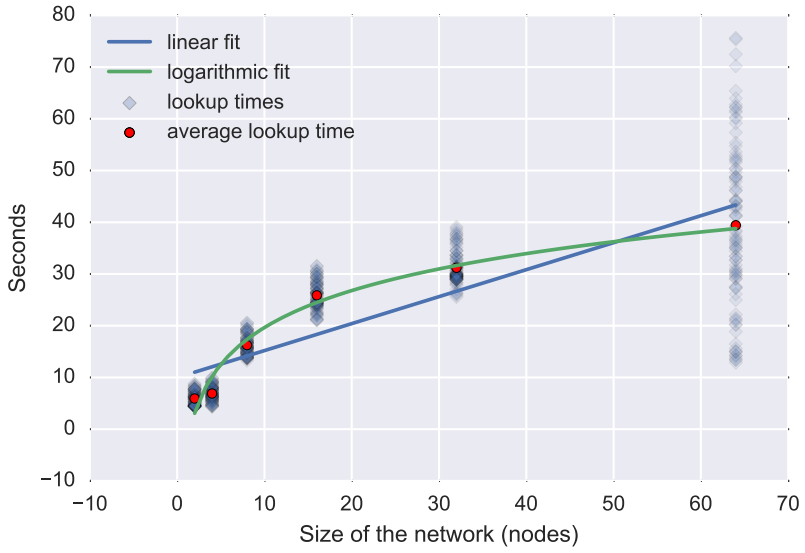


(b) Showing all sampled values

Figure 8.7: Look-up times recorded on a RAW DHT node running a block chain module *thick* node. Both plots show the linear and logarithmic fit to the data. (a) only shows mean values, while (b) shows also all the sampled values.



(a) Showing only average values



(b) Showing all sampled values

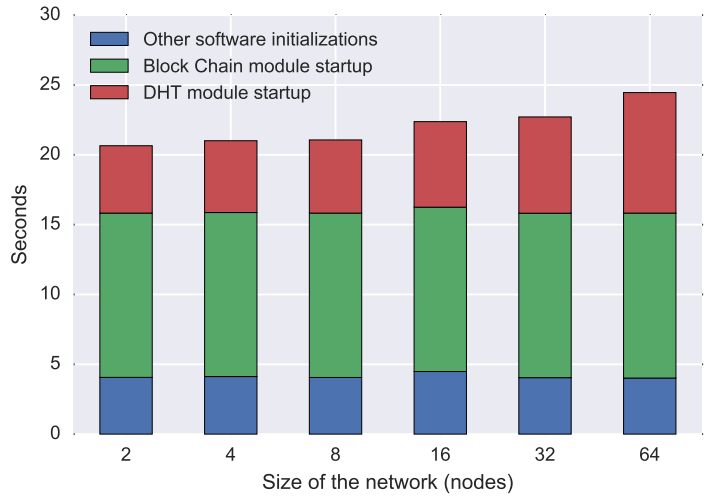
FIGURE 8.8: Look-up times recorded on a RAW DHT node running a block chain module *thin* node. Both plots show the linear and logarithmic fit to the data. (a) only shows mean values, while (b) shows also all the sampled values.

from the network. For each set was checked that

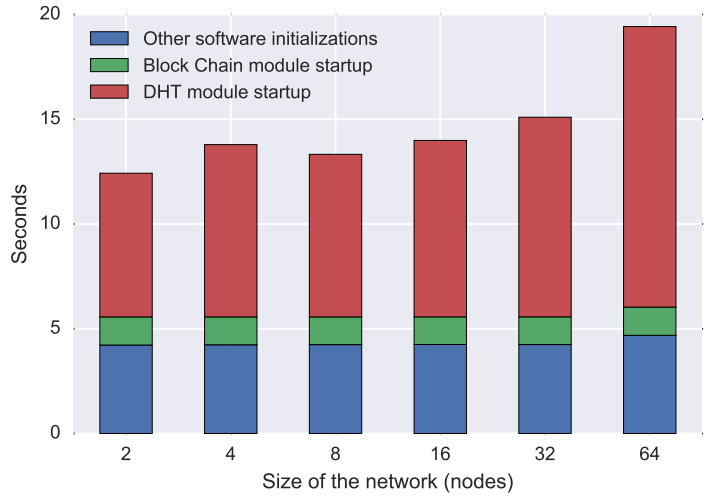
1. the searched key returned a value – nodes targeted are correct
2. the correct value is retrieved – targeted nodes does not store by chance a different value for the same key replica, as a possible residual of a previous unrelated store.

If both these conditions was true the operation was recorded as a success, otherwise as a failure. Look-up and find values resulted in a 100% success rate, regardless of network size and of the kind of local block chain client (*thick* or *thin*). This is of course the desirable behavior of any DHT system. However in a "general" DHT system keys' are positioned onto the address space following a fixed scheme, thus a search failure is always ascribable to nodes' failure. On the contrary, in an enhanced DHT like the one presented by Chapter 4 keys are expected to periodically rotate and change the set of nodes to whom they are entrusted to. RAW DHT complete accuracy performing look-up and find value operations proves that, within the same "seed epoch", keys' rotation mechanism is robust and reliable.

Throughout the experiments, occurrence time of a number of events was recorded on each node – e.g. start or finish time of certain software threads; finish time of objects' construction or initialization; ecc. These data was then aggregated measuring start-up times of the block chain and DHT module as well as the RAW DHT software as a whole. Figure 8.9 shows the average start-up times for nodes equipped with *thick* or *thin* block chain clients. The time spent initializing data structures and starting the block chain module is, as expected, essentially constant regardless of the joined network's size. Start-up time of the DHT module grows as the network grows. Albeit this is not a desirable behavior, it is not unexpected but is a fixable flaw. In Section 7.3 was already discussed the reason behind this software behavior. At start-up time the routing table is populated: when inserting a node into a RAW DHT's routing table an *isOldWorker* check is triggered. When bootstrapping node's references may be retrieved through the network or from a local file persisting references of nodes already contacted during previous software executions. Due to *isOldWorker* checks the bigger the network, the more nodes contacts are used at bootstrap, the longer will be the start-up time of the DHT module.



(a) Data from nodes equipped with a *thick* block chain client



(b) Data from nodes equipped with a *thin* block chain client

FIGURE 8.9: RAW DHT average start-up times. Full bars represent the time from the moment the software is started to the moment the DHT client is ready to manage user requests (or ready as a back-end). Each bar is broken down in time spent starting the DHT module, the block chain module and initializing other software data structures e.g. database, logging facilities, and other transient Java objects.

	Only Thick nodes						Mixed block chain clients					
Maximum nodes in network	2	4	8	16	32	64	2	4	8	16	32	64
Average failed nodes over time (%)	55.92	60.90	47.12	28.92	29.42	20.44	65.47	69.19	48.87	30.90	22.09	18.67
Maximum number of failed nodes (%)	100	100	87.5	62.50	65.62	42.18	100	100	75.00	68.75	43.75	35.93

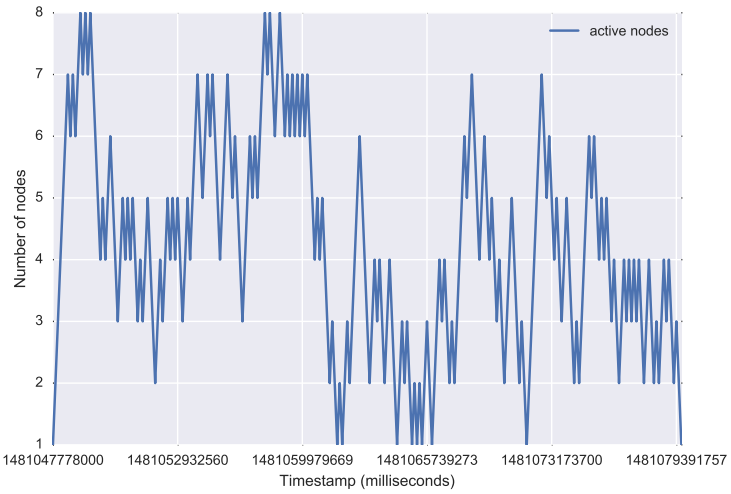
Table 8.4: This table shows, for every "churning network" experimental setup, both the average percentage of nodes that failed during the distributed software run and the maximum percentage of nodes failing during any of such experiments. Note that during each of the experiments the network evolved (nodes leaving or re-joining it) randomly.

Clearly nodes loaded from persisted files has already undergo an *isOldWorker* check before being persisted and may still be valid instances. So, as previously discussed in Section 7.3, upgrading RAW DHT software is expected to improve (even if mildly) the performance during the bootstrap phase.

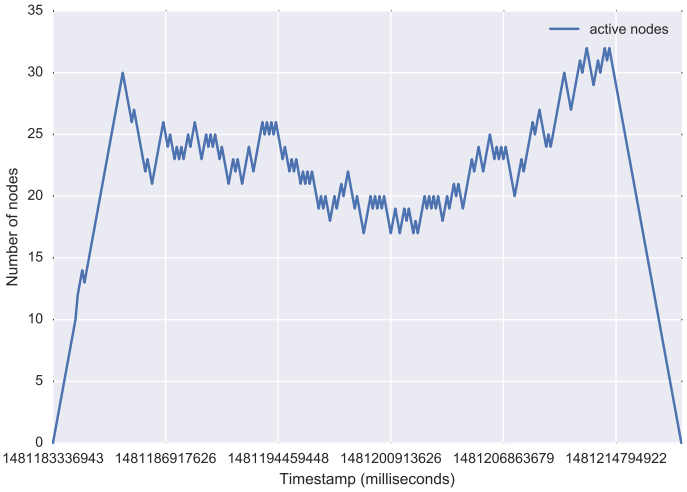
To confirm data gathered from the experiments discussed above, a set of experiments involving churning networks was performed. This set of measurements was carried out to document the (expected) degradation of performance caused by nodes joining or leaving the network. Experiments was set up in similar fashion to those already described: networks' sizes used as test-bed was power of two from 2 to 64, half of the experiments performed using only RAW DHT nodes equipped with *thick* block chain clients and half of the experiments used by both *thick* and *thin* nodes.

All these experiments was performed recording times on each node partaking to the network. Each node involved queried the network to store key/value records. Differently from experiments that lead to previously shown data, at any time each of the RAW DHT nodes could leave the network or join it anew. Figure 8.10 shows, as an example, the number of active nodes involved during two of the performed experiments. For each experiment Table 8.4 describes (as percentage) both the maximum number of nodes failing during each experiment, and the average number of inactive nodes relatively to the maximum allowed nodes.

It should be obvious that, due to the changes occurring in the whole network, routing tables held by working nodes are expected to progressively degrade, as soon as contained entries becomes outdated. Routing table degrade eventually leads to longer look-ups and store procedures, being the first hop(s) more likely to result in a failure. Longer routes clearly will end up resulting in longer look-up and store times: Figure 8.11 shows the average storage times recorded in function of the network size. As can be seen RAW DHT performance degrade when churn enters the picture. However it is reassuring that performance degradation appears to be reasonable. Moreover degradation appears to be stronger in smaller networks: failure of a single node in a small network is much more likely to (negatively) influence operations of the other nodes, while in a bigger network failure of a node impacts a small fraction of the active nodes. As a final consideration even if performance degraded the software continued to correctly carry out its functionality, in



(a) 8 maximum number of nodes allowed



(b) 32 maximum number of nodes allowed

Figure 8.10: Plot showing the number of of active nodes during evolution of a network made up of a maximum number of 8 or 32 nodes, each of them equipped with a *thick* block chain client. The two plots shows different rates of nodes reshuffling.

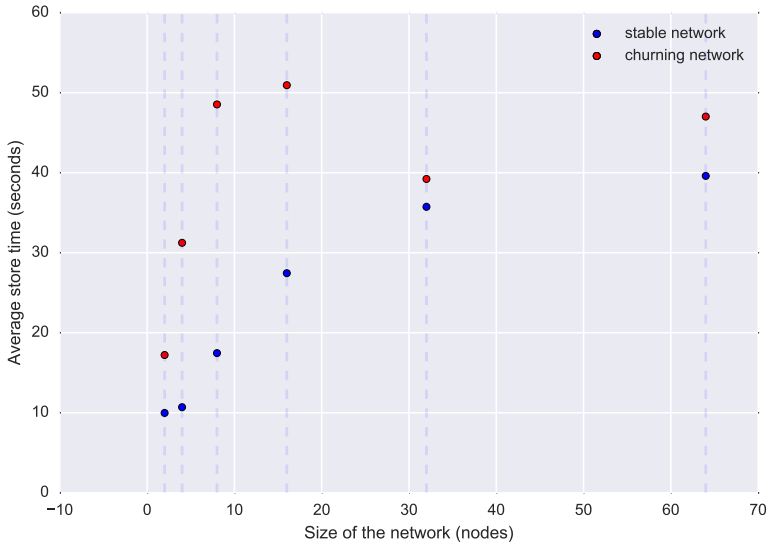


FIGURE 8.11: The red points mark the average store times recorded during the churning networks experiments. Blue points instead marks the average store times (both from Figure 8.4 and Figure 8.5). Note that the dashed vertical lines are drawn exclusively to guide the eye: particularly in the left side of the graph points coupling may be mistaken.

spite of node failures introduced by the experimental setup.

Recorded performance shows that RAW DHT is still a prototype and proof-of-concept software whose efficiency can (and should) be improved – as previously explained in Section 7.3. However looking at the results presented in previous paragraphs, by experimental evaluation RAW DHT software appears to be already a possible implementation of a secure DHT – fitting the one model outlined in Chapter 4. Moreover the system RAW DHT proves – directly on the field – the feasibility of the secure DHT design that this work presents. It must also be stressed out that the block chain module, although allowing the implementation of the new primitives that grant security to the whole system, is one of RAW DHT current performance bottlenecks and can be replaced by any other mechanism, as long as *getSeed* and *isOldWorker* can be backed up.

Conclusions 9

With this dissertation we presented a novel Distributed Hash Table relying on primitives based on the three ingredients of randomness, age and work. Our new design can be adopted modifying any existent DHT that satisfies some very mild and "natural" requirements. It is resilient against *Node insertion attacks* and mitigates appearance of *Sybil* nodes. Alongside the theoretical model of our new and secure DHT we also developed a working implementation, to corroborate its validity.

We provided a general description of DHT systems and supplied motivation for their possible usage. Albeit the various DHTs proposed over the years differ from one another in many ways, the basic model outlined in Chapter 2 is a valid representation of virtually all presented in the literature. In the same Chapter we also gave details on their various implementations. It is important to note that all systems described by our basic DHT abstraction may be undermined exploiting the same weak spots: in fact *Distributed Hash Tables* (DHTs) are known to be vulnerable to a number of security threats described in Chapter 3.

Our system was developed with a specific purpose: guarantee some form of security for stored DHT records against fraudulent deletion or tampering – i.e. against the *Node insertion attack*. Leveraging on the idea that an attacker must carefully position its nodes to gain control on target resources, we elaborated on the intuition that a possible solutions was as simple as periodi-

cally *moving* the resources – i.e. "resource rotation". This led to a system based on three principles: randomness – to guarantee unpredictability of resource rotation; age – to guarantee that a node chose its position before last resource rotation occurred; and work – to mitigate inconsiderate spawning of node identifiers. More specifically these three principles translate in two new primitives: *getSeed* and *isOldWorker*. A new DHT design based on the two new primitives is detailed in Chapter 4

We gave a theoretical background to our new DHT design. In Chapter 5 we formalized *getSeed* and *isOldWorker* primitives and proved that our proposed mechanism is correct and can protect the system against a malicious adversary. We also pointed out a that an additional layer of security can be obtained by prudently populating the *routing tables*. These results strengthen our confidence in the enhanced DHT.

To confirm practical feasibility of the new DHT enhanced model we developed it into a functional software. We implemented RAW DHT in the form of a Kademlia-based Java library, detailed in Chapter 7. Our system has two primary modules: the DHT one and a customized block chain client. The first provides a front-end accessing common DHT features – e.g. *look-ups* or *find values*. The latter is a block-chain-based back-end used by the core DHT module to retrieve a shared random seed and to timestamp IDs. In Chapter 6 we described block chains in general and, elaborating on them, we devised a way to repurpose them from a cryptocurrency system to a system that efficiently provides both the *getSeed* and *isOldWorker* primitives. RAW DHT's back-end was then developed as an actual implementation of the scheme presented in Section 6.2. It is crucial to note that our block chain module can be replaced with little effort in the future by any other mechanism providing the two primitives – e.g. one might imagine supporting *getSeed*, instead, from the entropy inherent in public stock-market oscillations.

The software was tested in a laboratory network demonstrating that a system relying on an hybrid block chain / DHT is actually usable. We tested RAW DHT for responsiveness and accuracy simulating different usage scenarios – see Chapter 8. Experiments proved it to be a perfectible yet already usable library. Thus it can be considered an answer to those circumstances requiring dynamic resources location with security guarantees on records availability.

More importantly, as already mentioned, the novel DHT design

we propose can be adopted independently of our proof-of-concept implementation. It is sufficiently general: with enough ingenuity it can be adapted to any existent DHT proposed either in literature or practice. Thus our idea encompass a broad spectrum of applications, delivering an elegant foundation for resilient overlays.

Summarizing, this dissertation fills a gap in Distributed Hash Table's security by presenting a blueprint for a Distributed Hash Table capable of safeguarding accessibility to its own records, implemented in a proof-of-concept yet fully operational *Random-Age-Work DHT* (RAW DHT).

Bibliography

[1] Sohail Abbas, Madjid Merabti, David Llewellyn-Jones, and Kashif Kifayat. "Lightweight sybil attack detection in MANETs". In: *IEEE Systems Journal* 7.2 (2013), pp. 236–248. ISSN: 19328184. DOI: 10.1109/JSYST.2012.2221912.

[2] Mehmud Abliz and Taieb Znati. "A Guided Tour Puzzle for Denial of Service Prevention". In: *2009 Annual Computer Security Applications Conference*. IEEE, Dec. 2009, pp. 279–288. ISBN: 978-1-4244-5327-6. DOI: 10.1109/ACSAC.2009.33. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5380686.

[3] Ruj Akavipat, Mahdi N. Al-Ameen, Apu Kapadia, Zahid Rahman, Roman Schlegel, and Matthew Wright. "ReDS: A Framework for Reputation-Enhanced DHTs". In: *IEEE Transactions on Parallel and Distributed Systems* 25.2 (Feb. 2014), pp. 321–331. ISSN: 1045-9219. DOI: 10.1109/TPDS.2013.231. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6594735.

[4] Lorenzo Alvisi, Allen Clement, Alessandro Epasto, Silvio Lattanzi, and Alessandro Panconesi. "SoK: The Evolution of Sybil Defense via Social Networks". In: *2013 IEEE Symposium on Security and Privacy*. 2. IEEE, May 2013, pp. 382–396. ISBN: 978-0-7695-4977-4. DOI: 10.1109/SP.2013.33. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6547122.

[5] Mahdi Nasrullah Al-Ameen and Matthew Wright. "Design and evaluation of persea, a sybil-resistant DHT". In: *Proceedings of the 9th ACM symposium on Information, computer and communications security - ASIA CCS '14*. New York, New York, USA: ACM Press, 2014, pp. 75–86. ISBN:

9781450328005. DOI: 10.1145/2590296.2590326. URL: http://dl.acm.org/citation.cfm?doid=2590296.2590326.

[6] Mahdi Nasrullah Al-Ameen and Matthew Wright. "iPersea: Towards improving the Sybil-resilience of social DHT". In: *Journal of Network and Computer Applications* 71 (Aug. 2016), pp. 1–10. ISSN: 10848045. DOI: 10.1016/j.jnca.2016.05.014. URL: http://linkinghub.elsevier.com/retrieve/pii/S1084804516301096.

[7] Jean Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. "BLAKE2: Simpler, smaller, fast as MD5". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7954 LNCS (2013), pp. 119–135. ISSN: 03029743. DOI: 10.1007/978-3-642-38980-1_8.

[8] Baruch Awerbuch and Christian Scheideler. "Towards a Scalable and Robust DHT". In: *Theory of Computing Systems* 45.2 (Aug. 2009), pp. 234–260. ISSN: 1432-4350. DOI: 10.1007/s00224-008-9099-9. URL: http://link.springer.com/10.1007/s00224-008-9099-9.

[9] Adam Back. *Hashcash - A Denial of Service Counter-Measure*. Tech. rep. August. 2002, pp. 1–10. URL: http://www.hashcash.org/Papers/Hashcash.Pdf.

[10] Leyla Bilge, Thorsten Strufe, Davide Balzarotti, and Engin Kirda. "All your contacts are belong to us: Automated Identity Theft Attacks on Social Networks". In: *Proceedings of the 18th international conference on World wide web - WWW '09*. New York, New York, USA: ACM Press, 2009, p. 551. ISBN: 9781605584874. DOI: 10.1145/1526709.1526784. URL: http://portal.acm.org/citation.cfm?doid=1526709.1526784.

[11] Nikita Borisov. "Computational Puzzles as Sybil Defenses". In: *Sixth IEEE International Conference on Peer-to-Peer Computing (P2P'06)*. IEEE, 2006, pp. 171–176. ISBN: 0-7695-2679-9. DOI: 10.1109/P2P.2006.10. URL: http://ieeexplore.ieee.org/document/1698607/.

[12] Ahmet Burak Can and Bharat Bhargava. "SORT: A Self-ORganizing Trust Model for Peer-to-Peer Systems". In: *IEEE Transactions on Dependable and Secure Computing* 10.1 (Jan. 2013), pp. 14–27. ISSN: 1545-5971. DOI: 10.1109/TDSC.2012.74. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6280552.

[13] Bengt Carlsson and Rune Gustavsson. "The rise and fall of napster-an evolutionary approach". In: *Proceedings of the 6th International Computer Science Conference on Active Media Technology - ATM '01* 8 (2001), pp. 347–354. URL: http://link.springer.com/chapter/10.1007/3-540-45336-9_40.

[14] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S Wallach. "Secure routing for structured peer-to-peer overlay networks". In: *Proceedings of the 5th symposium on Operating systems design and implementation - OSDI '02*. Vol. 36. December. New York, New York, USA: ACM Press, 2002, p. 299. ISBN: 9781450301114. DOI: 10.1145/1060289.1060317. URL: http://portal.acm.org/citation.cfm?doid=1060289.1060317.

[15] David Chaum. "Blind Signatures for Untraceable Payments". In: *Advances in Cryptology*. Vol. 82. Boston, MA: Springer US, 1983, pp. 199–203. ISBN: 978-1-4757-0604-8, 978-1-4757-0602-4. DOI: 10.1007/978-1-4757-0602-4_18. URL: http://link.springer.com/10.1007/978-1-4757-0602-4_18.

[16] Alice Cheng and Eric Friedman. "Sybilproof reputation mechanisms". In: *Proceeding of the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems - P2PECON '05*. New York, New York, USA: ACM Press, 2005, p. 128. ISBN: 1595930264. DOI: 10.1145/1080192.1080202. URL: http://portal.acm.org/citation.cfm?doid=1080192.1080202.

[17] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. "Freenet: A Distributed Anonymous Information Storage and Retrieval System". In: *Designing Privacy Enhancing Technologies*. 2001, pp. 46–66. ISBN: 978-3-540-44702-3. DOI: 10.1007/3-540-44702-4_4. URL: http://link.springer.com/10.1007/3-540-44702-4_4.

[18] Tyson Condie, Varun Kacholia, Sriram Sankararaman, Joseph M Hellerstein, Petros Maniatis, and U C Berkeley. "Induced Churn as Shelter from Routing-Table Poisoning". In: *In Proc. 13th Annual Network and Distributed System Security Symposium (NDSS)*. 2006.

[19] Cristiano Costa and Jussara Almeida. "Reputation Systems for Fighting Pollution in Peer-to-Peer File Sharing Systems". In: *Seventh IEEE International Conference on Peer-to-Peer Computing (P2P 2007)*. IEEE, Sept. 2007, pp. 53–60. ISBN: 0-7695-2986-0. DOI: 10.1109/P2P.2007.15. URL: http://ieeexplore.ieee.org/document/4343464/.

[20] Anthony Cuthbertson. *Bitcoin now accepted by 100,000 merchants worldwide*. 2015. URL: http://www.ibtimes.co.uk/bitcoin-now-accepted-by-100000-merchants-worldwide-1486613.

[21] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, and Robert Morris. "Designing a DHT for Low Latency and High Throughput". In: *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation* 1 (2004), p. 7. ISSN: 00045411. URL: http://portal.acm.org/citation.cfm?id=1251182.

[22] George Danezis, Chris Lesniewski-Laas, M. Frans Kaashoek, and Ross Anderson. "Sybil-Resistant DHT Routing". In: *Computer Security – ESORICS 2005*. Vol. 3679 LNCS. June. 2005, pp. 305–318. ISBN: 3540289631. DOI: 10.1007/11555827_18. URL: http://link.springer.com/10.1007/11555827_18.

[23] Quynh H. Dang. *Secure Hash Standard*. Tech. rep. October. Gaithersburg, MD: National Institute of Standards and Technology, July 2015, p. 36. DOI: 10.6028/NIST.FIPS.180-4. URL: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

[24] John R. Douceur. "The Sybil Attack". In: *Peer-to-peer Systems*. 2002, pp. 251–260. ISBN: 3540441794. DOI: 10.1007/3-540-45748-8_24. URL: http://link.springer.com/10.1007/3-540-45748-8_24.

[25] Evan Duffield and Daniel Diaz. *Dash: A PrivacyCentric CryptoCurrency*. Self-published, 2015. URL: https://www.dash.org/wp-content/uploads/2015/04/Dash-WhitepaperV1.pdf.

[26] Morris J. Dworkin. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. Tech. rep. August. Gaithersburg, MD: National Institute of Standards and Technology, July 2015. DOI: 10.6028/NIST.FIPS.202. URL: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[27] Amos Fiat and Jared Saia. "Censorship Resistant Peer-to-Peer Networks". In: *Theory of Computing* 3.1 (2007), pp. 1–23. ISSN: 1557-2862. DOI: 10.4086/toc.2007.v003a001. URL: http://www.theoryofcomputing.org/articles/v003a001.

[28] Amos Fiat, Jared Saia, and Maxwell Young. "Making Chord Robust to Byzantine Attacks". In: *Esa*. 191445. 2005, pp. 803–814. ISBN: 3-540-29118-0. DOI: 10.1007/11561071_71. URL: http://link.springer.com/10.1007/11561071_71.

[29] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. "Impossibility of Distributed Consensus with One Faulty Process". In: *Journal of the ACM* 32.2 (1985), pp. 374–382. ISSN: 00045411. DOI: 10.1145/3149.214121. URL: http://portal.acm.org/citation.cfm?doid=3149.214121.

[30] Gabriela Gheorghe, Renato Lo Cigno, and Alberto Montresor. "Security and privacy issues in P2P streaming systems: A survey". In: *Peer-to-Peer Networking and Applications* 4.2 (June 2011), pp. 75–91. ISSN: 1936-6442. DOI: 10.1007/s12083-010-0070-6. URL: http://link.springer.com/10.1007/s12083-010-0070-6.

[31] Stuart Haber and W.Scott Stornetta. "How to time-stamp a digital document". In: *Journal of Cryptology* 3.2 (1991), pp. 99–111. ISSN: 0933-2790. DOI: 10.1007/BF00196791. URL: http://link.springer.com/10.1007/BF00196791.

[32] Cyrus Harvesf and Douglas M. Blough. "The Effect of Replica Placement on Routing Robustness in Distributed Hash Tables". In: *Sixth IEEE International Conference on Peer-to-Peer Computing (P2P'06)*. IEEE, 2006, pp. 57–6. ISBN:

0-7695-2679-9. DOI: 10.1109/P2P.2006.44. URL: http://ieeexplore.ieee.org/document/1698591/.

[33] Mohammed Hawa, Loqman As-Sayid-Ahmad, and Loay D. Khalaf. "On enhancing reputation management using Peer-to-Peer interaction history". In: *Peer-to-Peer Networking and Applications* 6.1 (Mar. 2013), pp. 101–113. ISSN: 1936-6442. DOI: 10.1007/s12083-012-0142-x. URL: http://link.springer.com/10.1007/s12083-012-0142-x.

[34] Kirsten Hildrum and John Kubiatowicz. "Asymptotically Efficient Approaches to Fault-Tolerance in Peer-to-Peer Networks". In: *Distributed Computing*. 2003, pp. 321–336. ISBN: 978-3-540-39989-6. DOI: 10.1007/978-3-540-39989-6_23. URL: http://link.springer.com/10.1007/978-3-540-39989-6_23.

[35] Russell Housley. *Cryptographic Message Syntax (CMS)*. Tech. rep. Aug. 2002, pp. 1–57. DOI: 10.17487/rfc3369. URL: https://www.rfc-editor.org/info/rfc3369.

[36] Russell Housley, William Timothy Polk, Warwick Ford, and David Solo. *Internet X. 509 public key infrastructure certificate and certificate revocation list (CRL) profile*. Tech. rep. Apr. 2002, pp. 1–129. DOI: 10.17487/rfc3280. URL: https://www.rfc-editor.org/info/rfc3280.

[37] Yusuo Hu, Danqi Wang, Hui Zhong, and Feng Wu. "SocialTrust: Enabling long-term social cooperation in peer-to-peer services". In: *Peer-to-Peer Networking and Applications* 7.4 (Dec. 2014), pp. 525–538. ISSN: 1936-6442. DOI: 10.1007/s12083-013-0198-2. URL: http://link.springer.com/10.1007/s12083-013-0198-2.

[38] Gera Jaideep and Bhanu Prakash Battula. "Survey on the present state-of-the-art of P2P networks, their security issues and counter measures". In: *International Journal of Applied Engineering Research* 11.1 (2016), pp. 616–620. ISSN: 09739769.

[39] Markus Jakobsson and Ari Juels. "Proofs of Work and Bread Pudding Protocols(Extended Abstract)". In: *Secure Information Networks*. Boston, MA: Springer US, 1999, pp. 258–272. DOI: 10.1007/978-0-387-35568-9_18. URL: http://link.springer.com/10.1007/978-0-387-35568-9_18.

[40] Mian Ahmad Jan, Priyadarsi Nanda, Xiangjian He, and Ren Ping Liu. "A Sybil attack detection scheme for a forest wildfire monitoring application". In: *Future Generation Computer Systems* 1 (June 2016), pp. 1–14. ISSN: 0167739X. DOI: 10.1016/j.future.2016.05.034. URL: http://linkinghub.elsevier.com/retrieve/pii/S0167739X16301522.

[41] Oliver Jetter, Jochen Dinger, and Hannes Hartenstein. "Quantitative Analysis of the Sybil Attack and Effective Sybil Resistance in Peer-to-Peer Systems". In: *2010 IEEE International Conference on Communications*. IEEE, May 2010, pp. 1–6. ISBN: 978-1-4244-6402-9. DOI: 10.1109/ICC.2010.5501977. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5501977.

[42] Harry Kalodner, Miles Carlsten, Paul Ellenbogen, Joseph Bonneau, and Arvind Narayanan. "An empirical study of Namecoin and lessons for decentralized namespace design". In: *14th Annual Workshop on the Economics of Information Security (WEIS)*. 2015.

[43] P. Kavitha, C. Keerthana, V. Niroja, and V. Vivekanandhan. "Mobile-id Based Sybil Attack detection on the Mobile ADHOC Network 1". In: *International Journal of communication and computer Technologies* 02.02 (2014), pp. 6–9.

[44] Sunny King and Scott Nadal. *PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake*. Self-published, 2012. URL: http://ppcoin.org/static/ppcoin-paper.pdf.

[45] Dor Konforty, Yuval Adam, Daniel Estrada, and Lucius Gregory Meredith. *Synereo: The Decentralized and Distributed Social Network*. Self-published, 2015. URL: http://www.synereo.com/whitepapers/synereo.pdf.

[46] Jie Kong, Wandong Cai, and Lei Wang. "The Evaluation of Index Poisoning in BitTorrent". In: *2010 Second International Conference on Communication Software and Networks*. IEEE, 2010, pp. 382–386. ISBN: 978-1-4244-5726-7. DOI: 10.1109/ICCSN.2010.39. URL: http://ieeexplore.ieee.org/document/5437695/.

[47] P. Vinoth Kumar and M. Maheshwari. "Prevention of Sybil attack and priority batch verification in VANETs". In: *International Conference on Information Communication and Embedded Systems (ICICES2014)*. 978. IEEE, Feb. 2014, pp. 1–5. ISBN: 978-1-4799-3834-6. DOI: 10.1109/ICICES.2014.7033926. URL: http://ieeexplore.ieee.org/document/7033926/.

[48] Daniel Larimer, Ned Scott, Valentine Zavgorodnev, Benjamin Johnson, James Calfee, and Michael Vandeberg. *Steem An incentivized, blockchain-based social media platform*. March. Self-published, 2016. URL: https://steem.io/SteemWhitePaper.pdf.

[49] ByungKwan Lee, EunHee Jeong, and Ina Jung. "A DTSA (detection technique against a sybil attack) protocol using SKC (session key based certificate) on VANET". In: *International Journal of Security and its Applications* 7.3 (2013), pp. 1–10. ISSN: 17389976.

[50] Yeonju Lee, Hyelim Koo, Seungoh Choi, Byeong-hee Roh, and Cheolho Lee. "Advanced node insertion attack with availability falsification in Kademlia-based P2P networks". In: *14th International Conference on Advanced Communication Technology (ICACT)*. 2012, pp. 73–76. ISBN: 978-89-5519-163-9. URL: http://ieeexplore.ieee.org/document/6174613/.

[51] Chris Lesniewski-Laas. "A Sybil-proof one-hop DHT". In: *Proceedings of the 1st workshop on Social network systems - SocialNets '08*. New York, New York, USA: ACM Press, 2008, pp. 19–24. ISBN: 9781605581248. DOI: 10.1145/1435497.1435501. URL: http://portal.acm.org/citation.cfm?doid=1435497.1435501.

[52] Chris Lesniewski-Laas and M. Frans Kaashoek. "Whanau: A Sybil-proof Distributed Hash Table". In: *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*. NSDI'10. USENIX Association, 2010, pp. 111–126. ISBN: 978-931971-73-7. URL: http://www.usenix.org/events/nsdi10/tech/full_papers/lesniewski-laas.pdf.

[53] Frank Li, Prateek Mittal, Matthew Caesar, and Nikita Borisov. "SybilControl". In: *Proceedings of the seventh ACM workshop on Scalable trusted computing*. STC '12. New York, New York, USA: ACM Press, 2012, p. 67. ISBN: 978-1-4503-1662-0. DOI: 10.1145/2382536.2382548. URL: http://dl.acm.org/citation.cfm?doid=2382536.2382548.

[54] Jian Liang, Naoum Naoumov, and Keith W. Ross. "The Index Poisoning Attack in P2P File Sharing Systems". In: *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*. IEEE, 2006, pp. 1–12. ISBN: 1-4244-0221-2. DOI: 10.1109/INFOCOM.2006.232. URL: http://ieeexplore.ieee.org/document/4146885/.

[55] Thomas Locher, David Mysicka, Stefan Schmid, and Roger Wattenhofer. "Poisoning the Kad Network". In: *International Conference on Distributed Computing and Networking*. 2010, pp. 195–206. DOI: 10.1007/978-3-642-11322-2_22. URL: http://link.springer.com/10.1007/978-3-642-11322-2_22.

[56] Andrew Loewenstern and Arvid Norberg. *BitTorrent Enhancement Proposals 5: DHT Protocol*. Tech. rep. 2008. URL: http://www.bittorrent.org/beps/bep_0005.html.

[57] Andreas Loibl. "Namecoin". In: *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)*. August. 2014, pp. 107–113. DOI: 10.2313/NET-2014-08-1_14.

[58] Xiaosong Lou, Kai Hwang, and Yue Hu. "Accountable File Indexing against DDoS Attacks in Peer-to-Peer Networks". In: *GLOBECOM 2009 - 2009 IEEE Global Telecommunications Conference*. IEEE, Nov. 2009, pp. 1–6. ISBN: 978-1-4244-4148-8. DOI: 10.1109/GLOCOM.2009.5425979. URL: http://ieeexplore.ieee.org/document/5425979/.

[59] Leonardo Maccari, Matteo Rosi, Romano Fantacci, Luigi Chisci, Luca Maria Aiello, and Marco Milanesio. "Avoiding Eclipse Attacks on Kad/Kademlia: An Identity Based Approach". In: *2009 IEEE International Conference on Communications*. IEEE, June 2009, pp. 1–5. DOI: 10.1109/

ICC.2009.5198772. URL: http://ieeexplore.ieee.org/document/5198772/.

[60] Samuel Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. "TAG: A Tiny AGgregation Service for Ad-hoc Sensor Networks". In: *Proceedings of the 5th symposium on Operating systems design and implementation* 36.SI (2002), pp. 131–146. ISSN: 0163-5980. DOI: 10.1145/844128.844142. URL: http://doi.acm.org/10.1145/844128.844142.

[61] JunPeng Mao, YanLi Cui, JianHua Huang, and JianBiao Zhang. "Analysis of Pollution Disseminating Model of P2P Network". In: *2008 Second International Symposium on Intelligent Information Technology Application*. Vol. 3. IEEE, Dec. 2008, pp. 790–794. ISBN: 978-0-7695-3497-8. DOI: 10.1109/IITA.2008.405. URL: http://ieeexplore.ieee.org/document/4740105/.

[62] Sergio Marti, Prasanna Ganesan, and Hector Garcia-Molina. "DHT Routing Using Social Links". In: *Peer-To-Peer Systems Iii*. Vol. 3279. 2005, pp. 100–111. ISBN: 3-540-24252-X. DOI: 10.1007/978-3-540-30183-7_10. URL: http://link.springer.com/10.1007/978-3-540-30183-7_10.

[63] Petar Maymounkov and D Mazieres. "Kademlia: A peer-to-peer information system based on the xor metric". In: *First International Workshop on Peer-to-Peer Systems*. 2002, pp. 53–65. ISBN: 978-3-540-44179-3. DOI: 10.1007/3-540-45748-8. URL: http://link.springer.com/chapter/10.1007/3-540-45748-8_5.

[64] Ralph C. Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: *Crypto*. 1988, pp. 369–378. ISBN: 3540187960. DOI: 10.1007/3-540-48184-2_32. URL: http://link.springer.com/10.1007/3-540-48184-2_32.

[65] Bryan N. Mills and Taieb F. Znati. "SCAR - Scattering, Concealing and Recovering Data within a DHT". In: *41st Annual Simulation Symposium (anss-41 2008)*. IEEE, Apr. 2008, pp. 35–42. ISBN: 978-0-7695-3143-4. DOI: 10.1109/ANSS-41.2008.38. URL: http://ieeexplore.ieee.org/document/4494403/.

[66] Mthcl. *The math of Nxt forging*. 1. Self-published, 2014, pp. 1–32. URL: https://www.docdroid.net/e29h/forging0-5-1.pdf.html.

[67] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Self-published, 2008. URL: https://bitcoin.org/bitcoin.pdf.

[68] Moni Naor and U. Wieder. "A simple fault tolerant distributed hash table". In: *Peer-to-Peer Systems II* (2003), pp. 88–97. URL: http://www.springerlink.com/index/4e756fgyq4ff4kay.pdf.

[69] Yuusuke Ookita and Satoshi Fujita. "Cost-effective index poisoning scheme for P2P file sharing systems". In: *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*. IEEE, June 2016, pp. 1–6. ISBN: 978-1-5090-0806-3. DOI: 10.1109/ICIS.2016.7550732. URL: http://ieeexplore.ieee.org/document/7550732/.

[70] Soyoung Park, Baber Aslam, Damla Turgut, and Cliff C. Zou. "Defense against Sybil attack in the initial deployment stage of vehicular ad hoc network based on roadside unit support". In: *Security and Communication Networks* 6.4 (Apr. 2013), pp. 523–538. ISSN: 19390114. DOI: 10.1002/sec.679. URL: http://doi.wiley.com/10.1002/sec.679.

[71] Riccardo Pecori and Luca Veltri. "Trust-based routing for Kademlia in a sybil scenario". In: *2014 22nd International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. IEEE, Sept. 2014, pp. 279–283. ISBN: 978-9-5329-0052-1. DOI: 10.1109/SOFTCOM.2014.7039131. URL: http://ieeexplore.ieee.org/document/7039131/.

[72] Colin Percival. *Stronger key derivation via sequential memory-hard functions*. Self-published, 2009, pp. 1–16. URL: http://www.tarsnap.com/scrypt/scrypt.pdf.

[73] Pratama Putra and Akihiro Nakao. "An Effective Index Poisoning Algorithm for Controlling Peer-to-Peer Network Applications". In: *Proceedings of the 2011 International Workshop on Modeling, Analysis, and Control of Complex Networks*. CNET '11. 2011, pp. 17–22. ISBN: 978-0-9836283-1-6. URL: http://dl.acm.org/citation.cfm?id=2043530.

[74] Mina Rahbari and Mohammad Ali Jabreil Jamali. "Efficient Detection of Sybil attack Based on Cryptography in Vanet". In: *International Journal of Network Security & Its Applications* 3.6 (Nov. 2011), pp. 185–195. ISSN: 09752307. DOI: 10.5121/ijnsa.2011.3614. URL: http://www.airccse.org/journal/nsa/1111nsa14.pdf.

[75] Shanta Rangaswamy and Vinay Hegde. "A Survey of Techniques to Defend Against Sybil Attacks in Social Networks". In: *International Journal of Advanced Research in Computer and Communication Engineering* 3.5 (2014), pp. 6577–6580. ISSN: 2278-1021.

[76] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. "A scalable content-addressable network". In: *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications - SIGCOMM '01*. New York, New York, USA: ACM Press, 2001, pp. 161–172. ISBN: 1581134118. DOI: 10.1145/383059.383072. URL: http://portal.acm.org/citation.cfm?doid=383059.383072.

[77] Rodrigo Rodrigues and Barbara Liskov. "High Availability in DHTs: Erasure Coding vs. Replication". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 3640 LNCS. 2. 2005, pp. 226–239. ISBN: 3540290680. DOI: 10.1007/11558989_21. URL: http://link.springer.com/10.1007/11558989_21.

[78] Hosam Rowaihy, William Enck, Patrick McDaniel, and Thomas La Porta. "Limiting Sybil Attacks in Structured P2P Networks". In: *IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications*. NAS-TR-0017-2005. IEEE, 2007, pp. 2596–2600. ISBN: 1-4244-1047-9. DOI: 10.1109/INFCOM.2007.328. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4215910.

[79] Antony Rowstron and Peter Druschel. "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems". In: *Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms*

Heidelberg, Germany, November 12–16, 2001 Proceedings. Ed. by Rachid Guerraoui. November 2001. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 329–350. ISBN: 978-3-540-45518-9. DOI: 10.1007/3-540-45518-3_18. URL: http://link.springer.com/10.1007/3-540-45518-3_18.

[80] Rynomster and Tecnovert. *ShadowCash : Zero knowledge Anonymous Distributed E Cash via Traceable Ring Signatures*. Self-published, 2014. URL: http://bravenewcoin.com/assets/Whitepapers/ShadowCash-Zeroknowledge-Anonymous-Distributed-ECash.pdf.

[81] Nicolas van Saberhagen. *CryptoNote v 2.0*. Self-published, 2013, pp. 1–20. URL: https://cryptonote.org/whitepaper.pdf.

[82] Atui Singh, Tsuen-Wan Ngan, Peter Druschel, and Dan S. Wallach. "Eclipse Attacks on Overlay Networks: Threats and Defenses". In: *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*. IEEE, 2006, pp. 1–12. ISBN: 1-4244-0221-2. DOI: 10.1109/INFOCOM.2006.231. URL: http://ieeexplore.ieee.org/document/4146884/.

[83] Emil Sit and Robert Morris. "Security Considerations for Peer-to-Peer Distributed Hash Tables". In: *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*. 2002, pp. 261–269. DOI: 10.1007/3-540-45748-8_25. URL: http://link.springer.com/10.1007/3-540-45748-8_25.

[84] Moritz Steiner, Taoufik En-Najjary, and Ernst W. Biersack. "A global view of kad". In: *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement - IMC '07*. New York, New York, USA: ACM Press, 2007, p. 117. ISBN: 9781595939081. DOI: 10.1145/1298306.1298323. URL: http://portal.acm.org/citation.cfm?id=1298306.1298323.

[85] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. "Chord". In: *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications - SIGCOMM '01*. New York, New York, USA: ACM Press, 2001, pp. 149–

160. ISBN: 1581134118. DOI: 10.1145/383059.383071. URL: http://portal.acm.org/citation.cfm?doid=383059.383071.

[86] Florian Tegeler and Xiaoming Fu. "SybilConf: Computational Puzzles for Confining Sybil Attacks". In: *2010 INFOCOM IEEE Conference on Computer Communications Workshops*. IEEE, Mar. 2010, pp. 1–2. ISBN: 978-1-4244-6739-6. DOI: 10.1109/INFCOMW.2010.5466685. URL: http://ieeexplore.ieee.org/document/5466685/.

[87] Nguyen Tran, Jinyang Li, Lakshminarayanan Subramanian, and Sherman S.M. Chow. "Optimal Sybil-resilient node admission control". In: *2011 Proceedings IEEE INFOCOM*. IEEE, Apr. 2011, pp. 3218–3226. ISBN: 978-1-4244-9919-9. DOI: 10.1109/INFCOM.2011.5935171. URL: http://ieeexplore.ieee.org/document/5935171/.

[88] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. "A survey of DHT security techniques". In: *ACM Computing Surveys* 43.2 (Jan. 2011), pp. 1–49. ISSN: 03600300. DOI: 10.1145/1883612.1883615. URL: http://portal.acm.org/citation.cfm?doid=1883612.1883615.

[89] P. Raghu Vamsi and Krishna Kant. "Sybil attack detection using Sequential Hypothesis Testing in Wireless Sensor Networks". In: *2014 International Conference on Signal Propagation and Computer Technology (ICSPCT 2014)*. IEEE, July 2014, pp. 698–702. ISBN: 978-1-4799-3140-8. DOI: 10.1109/ICSPCT.2014.6884945. URL: http://ieeexplore.ieee.org/document/6884945/.

[90] Pavel Vasin. *BlackCoin's Proof-of-Stake Protocol v2*. Self-published, 2014. URL: https://blackcoin.co/blackcoin-pos-protocol-v2-whitepaper.pdf.

[91] Bimal Viswanath, Ansley Post, Krishna P. Gummadi, and Alan Mislove. "An analysis of social network-based Sybil defenses". In: *ACM SIGCOMM Computer Communication Review* 40.4 (Aug. 2010), p. 363. DOI: 10.1145/1851275.1851226. URL: http://dl.acm.org/citation.cfm?doid=1851275.1851226.

[92] Guojun Wang, Felix Musau, Song Guo, and Muhammad Bashir Abdullahi. "Neighbor Similarity Trust against Sybil Attack in P2P E-Commerce". In: *IEEE Transactions on Parallel and Distributed Systems* 26.3 (Mar. 2015), pp. 824–833. ISSN: 1045-9219. DOI: 10.1109/TPDS.2014.2312932. URL: http://ieeexplore.ieee.org/document/6776524/.

[93] Liang Wang and Jussi Kangasharju. "Measuring large-scale distributed systems: case of BitTorrent Mainline DHT". In: *IEEE P2P 2013 Proceedings*. IEEE, Sept. 2013, pp. 1–10. ISBN: 978-1-4799-0515-7. DOI: 10.1109/P2P.2013.6688697. URL: http://ieeexplore.ieee.org/document/6688697/.

[94] Liang Wang and Jussi Kangasharju. "Real-world sybil attacks in BitTorrent mainline DHT". In: *2012 IEEE Global Communications Conference (GLOBECOM)*. IEEE, Dec. 2012, pp. 826–832. ISBN: 978-1-4673-0921-9. DOI: 10.1109/GLOCOM.2012.6503215. URL: http://ieeexplore.ieee.org/document/6503215/.

[95] Peng Wang, Nicholas Hopper, Ivan Osipkov, and Yongdae Kim. *Myrmic: Secure and Robust DHT Routing*. Tech. rep. University of Minnesota at Twin Cities, Minneapolis/St. Paul, MN, 2007. URL: http://www-users.cs.umn.edu/~kyd/doc/wohk07.pdf.

[96] Yu Yang and Lan Yang. "A Survey of Peer-to-Peer Attacks and Counter Attacks". In: *Proceedings of the International Conference on Security and Management (SAM)*. 2011, p. 1.

[97] Haifeng Yu, Phillip B. Gibbons, Michael Kaminsky, and Feng Xiao. "SybilLimit: A Near-Optimal Social Network Defense Against Sybil Attacks". In: *IEEE/ACM Transactions on Networking* 18.3 (June 2010), pp. 885–898. ISSN: 1063-6692. DOI: 10.1109/TNET.2009.2034047. URL: http://ieeexplore.ieee.org/document/5313843/.

[98] Haifeng Yu, Michael Kaminsky, Phillip B. Gibbons, and Abraham Flaxman. "SybilGuard". In: *ACM SIGCOMM Computer Communication Review* 36.4 (Aug. 2006), p. 267. ISSN: 01464833. DOI: 10.1145/1151659.1159945. URL: http://dl.acm.org/citation.cfm?id=1159945.

[99] Quan Yuan, Aaron Little, Maggie Kabore, and Youssouf Kabore. "A Study of Index Poisoning in Peer-To-Peer File Sharing Systems". In: *International Journal on Cybernetics & Informatics* 3.6 (Dec. 2014), pp. 11–24. ISSN: 23208430. DOI: 10.5121/ijci.2014.3602. URL: http://airccse.org/journal/ijci/papers/3614ijci02.pdf.

[100] Kuan Zhang, Xiaohui Liang, Rongxing Lu, Kan Yang, and Xuemin Sherman Shen. "Exploiting mobile social behaviors for Sybil detection". In: *2015 IEEE Conference on Computer Communications (INFOCOM)*. Vol. 26. IEEE, Apr. 2015, pp. 271–279. ISBN: 978-1-4799-8381-0. DOI: 10.1109/INFOCOM.2015.7218391. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7218391.

[101] Tao Zhang, Li Lin, and Jian-biao Zhang. "Resource Reliability in Kad". In: *2011 Second International Conference on Networking and Distributed Computing*. Vol. 5. IEEE, Sept. 2011, pp. 187–191. ISBN: 978-1-4577-0407-9. DOI: 10.1109/ICNDC.2011.45. URL: http://ieeexplore.ieee.org/document/6047132/.

[102] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. "Tapestry: A Resilient Global-Scale Overlay for Service Deployment". In: *IEEE Journal on Selected Areas in Communications* 22.1 (Jan. 2004), pp. 41–53. ISSN: 0733-8716. DOI: 10.1109/JSAC.2003.818784. URL: http://ieeexplore.ieee.org/document/1258114/.

[103] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. *Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing*. Tech. rep. April. Computer Science Division University of California, Berkeley, 2001.