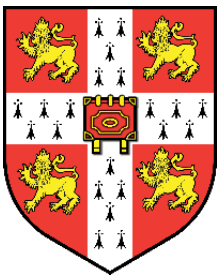
Towards a worldwide storage infrastructure

Julien Quintard
firstname.lastname@cl.cam.ac.uk

September 2010



University of Cambridge
Computer Laboratory

Jesus College

This dissertation is submitted for the degree of Doctor of Philosophy

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text.

This dissertation does not exceed the regulation length of sixty thousand words, including tables and footnotes.

Towards a worldwide storage infrastructure

Julien Quintard

Abstract

Peer-to-peer systems have recently gained a lot of attention in the academic community especially through the design of *KBR (Key-Based Routing)* algorithms and *DHT (Distributed Hash Table)*s. On top of these constructs were built promising applications such as video streaming applications but also storage infrastructures benefiting from the availability and resilience of such scalable network protocols.

Unfortunately, rare are the storage systems designed to be scalable and fault-tolerant to Byzantine behaviour, conditions required for such systems to be deployed in an environment such as the *Internet*. Furthermore, although some means of access control are often provided, such file systems fail to offer the end-users the flexibility required in order to easily manage the permissions granted to potentially hundreds or thousands of end-users. In addition, as for centralised file systems which rely on a special user, referred to as *root* on *Unices*, distributed file systems equally require some tasks to operate at the system level. The decentralised nature of these systems renders impossible the use of a single authoritative entity for performing such tasks since implicitly granting her superprivileges, unacceptable configuration for such decentralised systems.

This thesis addresses both issues by providing the file system objects a completely decentralised access control and administration scheme enabling users to express access control rules in a flexible way but also to request administrative tasks without the need for a superuser. A prototype has been developed and evaluated, proving feasible the deployment of such a decentralised file system in large-scale and untrustworthy environments.

Acknowledgments

I would like to thank Jean Bacon for her incredible patience, understanding and kindness. I am also indebted to Alastair Beresford for his advice and encouragement throughout the years. I would also like to thank the *Opera* group, especially Pedro Brandão, David Eyers, David Evans, Luis Vargas, Samuel Kounev, Jatinder Singh, Eiko Yoneki, Sriram Srinivasan, David Ingram, Salman Taherian, Scarlet Schwiderski and Ken Moody.

During my PhD, I have been fortunate enough to cross Myoung Jin Nam's path who I would like to thank sincerely for everything.

Finally, I would like to thank my parents and my friends for bearing with me all these years and making me a better person.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Contributions 3

1.3 Outline 5

2 Background **6**

2.1 Overlay Networks 8

2.1.1 Unstructured Overlay Networks 8

2.1.2 Hybrid Overlay Networks 10

2.1.3 Structured Overlay Networks 11

2.1.3.1 Chord 14

2.1.3.2 Kelips 16

2.1.4 Social Overlay Networks 18

2.2 Distributed Hash Tables 20

2.3 Peer-to-Peer File Systems 25

2.3.1 Pangaea 26

2.3.2 OceanStore 28

2.3.3 FARSITE 29

2.3.4 CFS 31

2.3.5 Ivy 33

2.3.6 Plutus 35

2.3.7 Pastis 37

3 Environment **39**

3.1 Properties . . . 39

3.2 Model . . . 43

3.2.1 File System . . . 43

3.2.2 Peer-to-Peer . . . 44

3.3 Mission . . . 45

3.4 Assumptions . . . 46

4 Design **53**

4.1 Access Control . . . 53

4.1.1 Objectives . . . 54

4.1.2 Model . . . 56

4.1.2.1 Policy . . . 56

4.1.2.2 Pattern . . . 57

4.1.2.3 Class . . . 57

4.1.3 Constraints . . . 59

4.1.4 Concept . . . 60

4.1.5 Scheme . . . 62

4.1.5.1 Physical Blocks . . . 62

4.1.5.2 Logical Blocks . . . 67

4.1.6 Algorithms . . . 75

4.1.7 Analysis . . . 80

4.2 Administration . . . 83

4.2.1 Semantics . . . 84

4.2.2 Model . . . 88

4.2.2.1 System-wide . . . 88

4.2.2.2 User-wide . . . 89

4.2.3 Objectives . . . 91

4.2.4 Scheme . . . 91

4.2.4.1 Community . . . 92

4.2.4.2 Ownership . . . 106

4.2.5 Algorithms . . . 108

4.2.6 Analysis . . . 116

5 Implementation **120**

5.1 Representation 120

5.2 Architecture 124

5.2.1 Elle 126

5.2.2 Lune 132

5.2.3 PIG 132

5.2.4 Agent 133

5.2.5 Etoile 135

5.2.6 Hole 140

6 Evaluation **145**

6.1 Methodology 146

6.1.1 Environments 146

6.1.2 Benchmarks 146

6.1.3 Metrics 152

6.2 Results 152

6.2.1 Overlay Network 152

6.2.2 Distributed Hash Table 153

6.2.3 File System 158

6.2.3.1 Implementation 158

6.2.3.2 Design 163

7 Conclusion **169**

Chapter 1

Introduction

This chapter introduces the thesis by detailing the motivation driving this research before presenting the contributions this work brings to the research community.

1.1 Motivation

Over the last decade, computers have become the universal tool for work, communication and entertainment. Despite the incredible technology progress, computers still fail to provide the end-user a way to deal with data in an easy, reliable and secure way. Although people use computers daily for both personal and professional tasks, users cannot rely on them when it comes to reliably storing documents, transparently sharing files with other users or synchronising data between multiple devices.

The following further details these three functionalities—storage, sharing and synchronisation—and explains why existing products and services fail to provide end-users the features and properties they expect.

Storage

Computer networks are growing rapidly in importance as a medium for the storage and exchange of information. After years of encouraging people to amass a hoard of digital media as well as to store personal data on their local hard disk, users now expect computers to become as reliable as any other home devices such as televisions, *Compact Disc* players and so forth.

Although computers will probably never be as reliable as televisions, most people feel like their local hard disk is a safe place for storing their sensitive files. The very few end-users concerned about losing their files tend to rely on manual backups.

Unfortunately, even for those users, files cannot be considered safe on external backups. Indeed, many plausible scenarios might lead to the complete loss of data including fire and theft amongst many others.

The *Internet* made it possible to store files on company-run storage clusters such as *Amazon*'s [DHJ+07] and many others [Ope, Omn, Box, Dro]. Unfortunately, people willing to use such services must completely trust the company for reliably storing their files while not disclosing or using their personal information for any purpose. For the obvious above reasons, such systems may not be suitable for everyone.

Peer-to-peer file sharing applications have gained great popularity over the last decade as a way for users to share their files with the rest of the world in a completely decentralised manner. Although peer-to-peer applications are interesting for increasing privacy and anonymity in the sense that nobody has complete control over the system, such applications do not provide any guarantees in terms of persistence, availability and security. Therefore, such peer-to-peer applications cannot be used for reliably storing users' files.

Sharing

Peer-to-peer file sharing applications completely changed the users' day-to-day *Internet* experience. Indeed, people are now used to launching such an application whenever they want to listen to an unknown band's music, download the last episode of their favourite *TV* series, watch or re-watch a famous movie and so forth.

Unfortunately, the well-known *eDonkey* [HKLF+06], *Gnutella* [PSAS01], *Bittorrent* [Coh03] *etc.* still lack some fundamental functionalities. Indeed, although these applications are generally very efficient at downloading popular content, no availability guarantees are provided for rare files, making it problematic for users to locate them.

Additionally, these applications usually rely on a flat name space, making it complicated for users to look for a rare file whose name resembles another popular but completely different file.

Finally, such applications basically aim at providing users a way to share their files with the rest of the world. However, one could be interested in controlling who has access to the shared files. Unfortunately, none of the well-known peer-to-peer file sharing applications provides any access control mechanism.

Although some company-run storage systems [Omn, Dro, XDr, Box] provide such sharing capabilities, they cannot succeed in offering as much diverse content as peer-to-peer communities, not mentioning the cost in storage and bandwidth for such a company to provide this service.

Finally, popular social websites such as *Youtube*, *Flickr etc.* also provide sharing capabilities but those services target a single medium such as video, sound or image, forcing the user to deal with multiple accounts and interfaces.

Synchronisation

With the advent of ubiquitous and mobile computing, people start getting their hands on multiple devices, all with amazing computing capabilities. With so many devices each with its own storage, users are forced to manually synchronise their data so that they can access a file from different devices and locations.

Company-run products such as *Apple*'s *MobileMe* [App], *Windows Live SkyDrive* [Win] *etc.* make it easier for end-users to synchronise their devices through the use of an online storage space.

Unfortunately, the online storage capacity is generally limited in many ways: number of files, file size, storage capacity and so forth. Besides, people might be concerned about privacy when relying on a private company for storing their personal and/or professional files. Finally, such applications often target the products of the same company only, making it difficult, if not impossible, for users to change or even use them on other systems.

⋆ ⋆
⋆

Interestingly, the three scenarios above have three points in common. Firstly, the devices involved are connected to the *Internet* being mobile phones, netbooks, office computers *etc.* Secondly, all these devices embed an unreliable storage capacity that can be used for storing, sharing and synchronising data. Thirdly, all these tasks—storing, sharing and synchronising—are related to the common abstraction known as the *file*.

A unique system for storing, sharing and synchronising files, independently of their medium type, in a reliable, secure and transparent way would therefore make it easy for users to manage their data.

1.2 Contributions

The thesis of this dissertation is that a file system abstraction on top of a peer-to-peer network is a viable platform and the most cost-effective one for ensuring the

fundamental properties end-users expect when it comes to storing, sharing and synchronising their data. As detailed in *Section 3.1*, these properties include availability, integrity, durability, privacy and efficiency among others.

The first contribution of this work is the definition of the properties that are required for the system to provide the end-users the expected guarantees.

Several peer-to-peer file systems such as *Ivy* [MMGC02], *CFS* [DKK+01], *Pastis* [mBPS05] *etc.* were developed over the last decade. However, very few of them provided common file system features, such as an access control mechanism for instance, while administration in such systems was completely ignored by the research community. The second contribution of this dissertation is the design of an access control and administration scheme for decentralised untrustworthy environments making them suitable building blocks for peer-to-peer file systems.

The third and final contribution is the implementation of a complete working peer-to-peer file system prototype along with an extensive evaluation proving feasible the deployment of such a system to a large community of users.

Figure 1.1 illustrates a peer-to-peer network connecting nodes physically distributed throughout the world. The work presented in this document aims at building a storage infrastructure on top of such a network in order to ensure fundamental properties such as reliability, availability, privacy, anonymity and so forth.

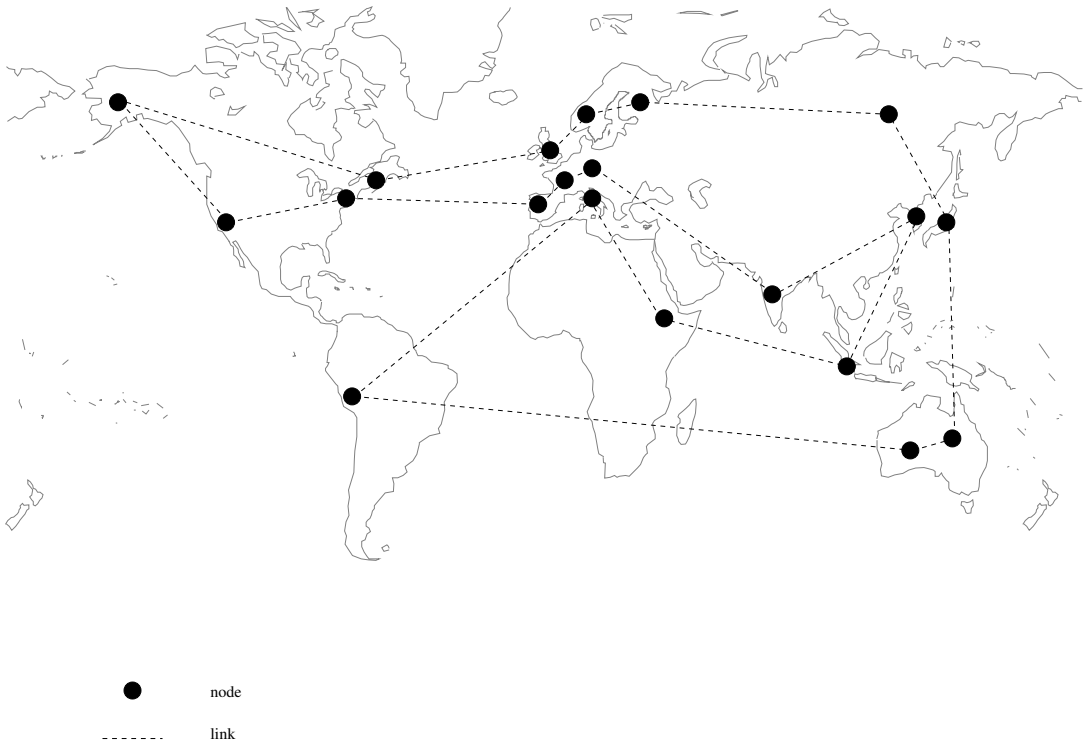


Figure 1.1: A worldwide storage infrastructure

1.3 Outline

The remainder of this dissertation is structured as follows.

Chapter 2 discusses the relevant background from overlay networks to distributed hash tables. Special attention is given to the presentation of the extensive body of work on peer-to-peer file systems.

Chapter 3 discusses the objectives of this work by precisely defining the required properties and deducing that the peer-to-peer file system model is suitable for achieving them all.

Chapter 4 discusses the semantic differences between centralised file systems and decentralised file systems and the impact on the user experience. Then, the chapter presents the design of the two building blocks peer-to-peer file systems require, namely, an access control and administration scheme.

The prototype implementation is discussed in *Chapter 5*, detailing how the system has been broken into small independent units and how they relate to each other.

Chapter 6 evaluates the performance of the prototype and validates the overall design. The chapter also suggests some possible improvements in specific areas.

Finally, *Chapter 7* concludes and discusses directions for future investigation.

Chapter 2

Background

This chapter introduces peer-to-peer systems from overlay networks to routing algorithms capable of locating a node given its identifier in a decentralised manner to distributed hash tables which provide a storage abstraction to peer-to-peer file systems which enable the user to interact with the system following a standard file system interface.

⋆ ⋆
⋆

Peer-to-peer systems differ from common distributed systems in the sense that nodes composing the network can self-organise with very little information on the whole network. Such networks are designed with fault tolerance in mind because the number of nodes populating such networks is generally so high that nodes disconnecting, crashing or acting maliciously are more probable than in other, more controlled, distributed systems.

Such systems are often used to aggregate the resources of many heterogeneous computers across the world. Although those resources can be very diverse, this document focuses on the storage capacity such nodes provide.

The lack of centralised servers makes such networks suitable to accommodate a very large number of nodes. However, these peer-to-peer networks also exhibit specific characteristics that need to be taken into account.

Scalability

The decentralised nature of peer-to-peer networks implies that the more nodes join the network, the more aggregated resources the system acquires, hence, the better the system.

However, the network must cope with this potentially large number of nodes by relying on scalable protocols ensuring that the system keeps providing client nodes the expected service as nodes dynamically join and leave the network.

Latency

Unlike centralised topologies that require low-latency servers with a high bandwidth to supply all the clients, peer-to-peer networks rely mostly on personal computers. Such computers are generally connected to the network through a high-latency and low-bandwidth *Internet* connection.

Systems built upon such networks, *e.g. Internet*, therefore cannot afford using the same protocols and algorithms as for centralised or partially-distributed systems.

Churn

The decentralised nature of peer-to-peer networks implies that every node contributes to the system by taking part in the basic tasks such as routing messages between nodes, managing the network state *etc.*

Therefore, every node is considered an important component of the system. Whenever a node fails, other nodes must be informed and past operations involving this failing node may have to be re-performed. In addition, most peer-to-peer systems are open such that new nodes constantly join the network, in which case, the other nodes must be informed of their arrival.

Unfortunately, studies showed that the churn rate of the studied peer-to-peer networks was high [LSG+04, RGRK04]. Peer-to-peer systems must integrate this characteristic in the design of their algorithms such that, for instance, nodes are not assumed to be connected to the network at all times.

Untrustworthiness

Clients composing a peer-to-peer network run on computers under the full control of their respective user. The system therefore has no authority to force nodes to follow the system's protocols. The network is thus assumed to be untrustworthy

since many of the nodes populating the system may be faulty. For example, a virus may have infected the whole client's operating system or the user may have installed a modified version to take advantage of the system without contributing resources, nodes referred to as *free rider*s.

Peer-to-peer systems must be designed with this property in mind making sure that nothing relies on a single node, such a node being faulty could endanger the system in its entirety.

2.1 Overlay Networks

The computers connected together and collaborating in the same peer-to-peer system form an *overlay network* [KS10] on top of a physical network *e.g.* the *Internet*.

The topology of the overlay network, its degree of decentralisation as well as the communication protocol, vary from one peer-to-peer system to another. These characteristics are fundamental as they impact the scalability and performance of the network but also its capacity to self-organise and tolerate faults.

Overlay networks can be classified in four categories according to the way nodes are connected to one another. Depending on the overlay network's topology, it may be easier to join/leave the network but more difficult to locate a precise object in the network.

The following discusses the different models of overlay networks along with the way objects are located in such networks.

2.1.1 Unstructured Overlay Networks

The very first deployed peer-to-peer applications enabled users to contribute files to the system that any other user could download. These peer-to-peer file sharing applications allow users to search for files matching the keywords the user specified. The objective of such a system is to locate all the files whose name matches with those keywords. Then, the user, through the application, can download the files of interest to her.

The overlay networks on which such applications were built had the property of lacking organisation in the way nodes were connected to each other. Besides, nodes connected to the network were all considered equal *i.e.* no node had more privileges than others. Such unstructured overlay networks [CFK03] are therefore sometimes referred to as being *flat*, forming a completely random graph. In such an environment, a node wishing to join the network basically has to connect to an already connected node.

Since such networks have no overall structure, unstructured overlay networks are very easy to manage. Indeed, whenever a node leaves the network, only its neighbours must detect its departure and update their internal state. However, the other nodes of the network do not need to be notified of the change in the network's topology, hence lowering the communication costs of maintenance.

Figure 2.1 depicts such a unstructured overlay network in which nodes are connected without following any pre-defined structure.

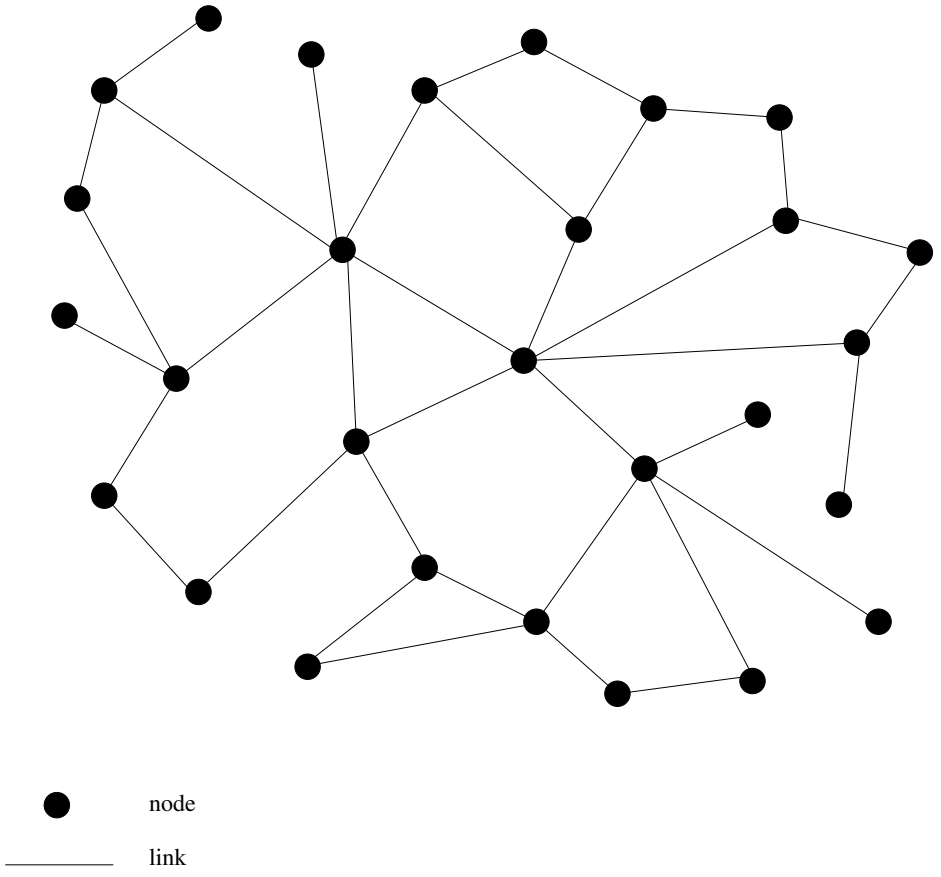


Figure 2.1: A flat unstructured overlay network

Locating an object—*e.g.* a file—in such a network without any centralised entity maintaining a global state of the network requires every node to contribute. Indeed, one of the first routing algorithms designed for unstructured overlay networks consisted of flooding the network. The node issuing the search request starts by sending a message to all its neighbour nodes, asking them to locate files matching a list of keywords. Whenever a node receives such a request, it starts by checking if it does have such files among the files it contributed to the peer-to-peer system, and replies to the requesting node accordingly. Then, the message is forwarded to all the other neighbours until the message expires *i.e.* the *TTL (Time To Live)* reaches zero.

Unfortunately, such an algorithm implies a high network overhead since messages

are sent to a large fraction of nodes which do not have the sought resource and are therefore not interested in the process. Such a routing algorithm and its variants [DHA03, YM02, YVGM04] are extremely simple to deploy and do not constrain the overlay network topology. However, the implied overhead makes these algorithms only suitable for small networks, though many projects are known to have used and still use them, most notably *Gnutella* [PSAS01] and *Freenet* [CSWH01] among many others [DFM01, DMS04].

2.1.2 Hybrid Overlay Networks

Although flat unstructured overlay networks are very good for handling churn, they do not perform well when it comes to locating a particular object or node. Hybrid overlay networks [SBA03, CRB+03], also known as multi-level unstructured overlay networks, address this problem by adding a level of highly-available *supernode*s, *a.k.a.* *superpeer*s, forming a small inner overlay network. These supernodes are responsible for referencing the nodes connected to the network along with the objects they contribute to the system.

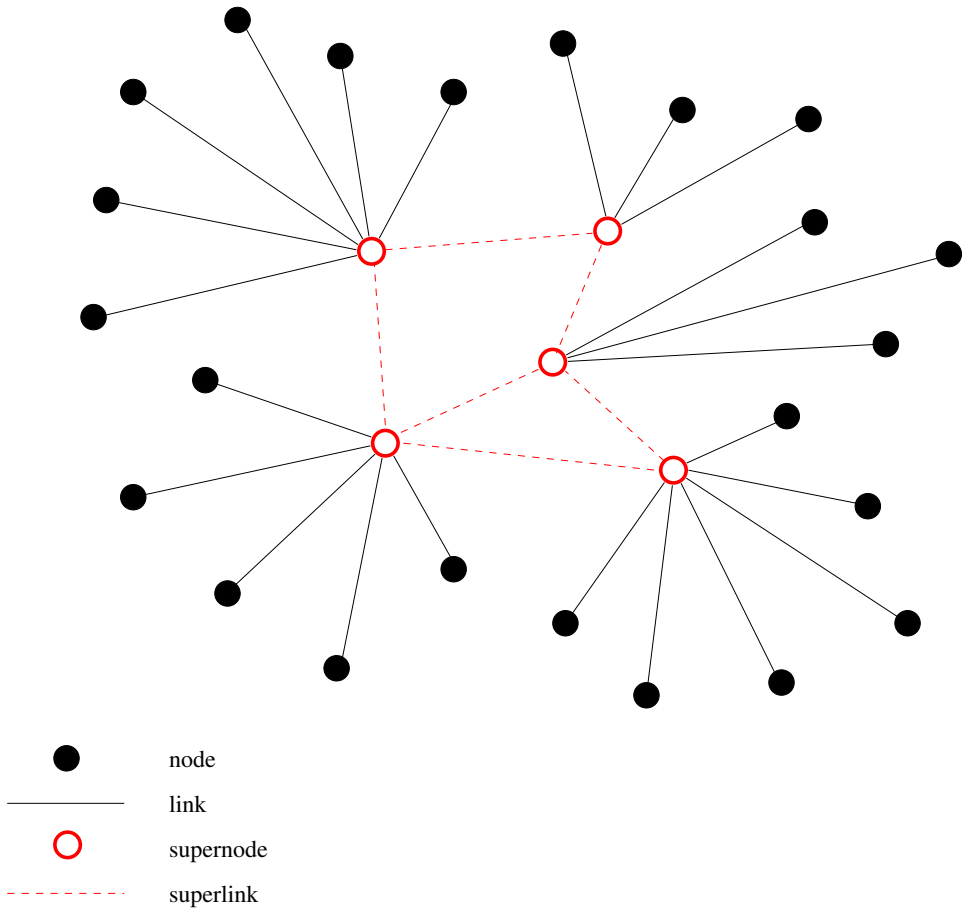


Figure 2.2: A two-level hybrid overlay network

The main drawback of such a topology is the high load implied as well as the large state that must be kept by the supernodes. *Figure 2.2* illustrates such a centralisation within the inner overlay of supernodes. If one of these supernodes fails, the impact on the overall network's performance may be disastrous as the load the faulty supernode was handling must be balanced on the others.

The routing algorithm in such overlay networks is however trivial. Indeed, whenever a user performs a search, the client node requests the supernode it is connected to, supplying some keywords. The supernode performs the matching process by comparing the keywords with the names of all the files in its records, and possibly contacts other supernodes if required.

Although such routing algorithms [GEvS07] involve only a few nodes, they require supernodes to be extremely reliable, powerful and well-connected in order to handle all the client nodes' requests.

2.1.3 Structured Overlay Networks

Structured overlay networks were developed to overcome the limitations of unstructured and hybrid overlay networks. Such networks are completely decentralised and organised such that nodes communicate with well-identified nodes according to the protocol in contrast with unstructured overlay networks in which nodes connect to other nodes in a unplanned way, hence forming a random graph.

Figure 2.3 illustrates a structured overlay network in which every node is assigned an identifier following a ring-based identifier space [PRR97]: nodes are identified by a number such that every node follows the node with the preceeding number—*i.e.* highest number which is smaller than the current node's—with the exception of the node with the smallest identifier which follows the node with the highest one, hence creating a loop within the identifier space.

Although unstructured and hybrid overlay networks were primarily used for keyword-based lookups, other search criteria such as object identifiers, regular expressions and so on could have been used. In contrast, structured overlay networks organise nodes by assigning them an identifier while routing algorithms make use of this organisation to perform fast lookups. Therefore, structured overlay networks were not designed to perform attribute-based lookups as quickly as identifier-based lookups, though some decentralised data structures [RH04] were designed for specific types of queries. Besides, dissemination techniques used in unstructured overlay networks can also [CCR05] be used in structured overlay networks. Routing algorithms based on identifiers are sometimes referred to as *KBR (Key-Based Routing)* algorithms and provide an interface composed of a single routine, `Lookup`(ι), which returns the *IP (Internet Protocol)* address of the node in charge of the identifier ι.

Although attribute-based routing algorithms used in unstructured and hybrid overlay networks enable rich searches within the set of objects, such algorithms do not scale well since they do not distribute the resource requirements evenly across the nodes. *KBR* algorithms, however, aim at locating the node responsible for an identifier. Such algorithms were designed to scale so that locating an identifier involves a small number of nodes while each node maintains only a few links to other nodes.

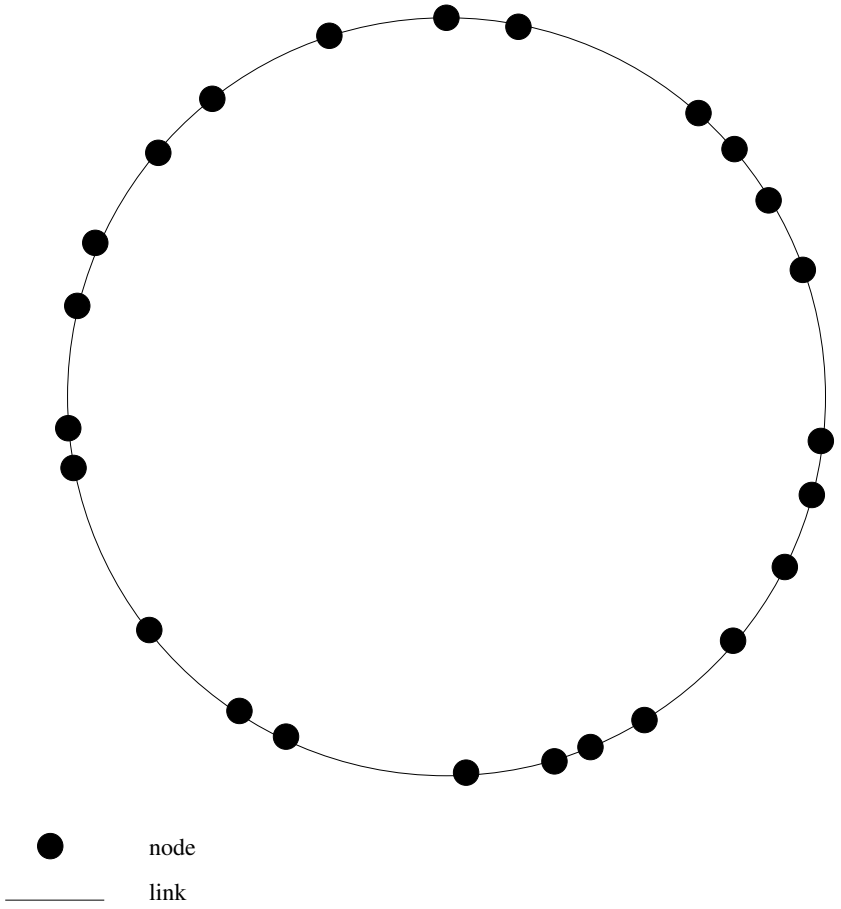


Figure 2.3: A ring-based structured overlay network

Every node in a structured overlay network is assigned an identifier from a large identifier space. Identifiers are generated in a random fashion in order to provide network resource balancing and fault tolerance. Besides, nodes with close or even adjacent identifiers are, with high probablity, in different geographic locations, under distinct users' control and with different computing and network resources.

Objects, *e.g.* data blocks, files *etc.*, are assigned identifiers from the same identifier space. Every object in the network is dynamically associated with a node, called the object's *home*, or sometimes *root*. This node is responsible for storing the object and answering requests related to this object.

Every node maintains a routing table containing the identifier and *IP* address of

some other nodes, depending on the topology. In most systems, nodes also maintain a set of neighbours containing the *IP* address of a few closest nodes. These two data structures are updated whenever a node is detected to have joined or left the network but also periodically in order to maintain the network in a consistent state.

KBR algorithms are distinctive from other routing algorithms in the way that they determine the size of the routing tables as well as the length of the search paths, as detailed next. These metrics are important as they characterise the robustness and performance of the routing algorithm, hence of the whole network. Indeed, the more entries in a routing table, the more communication is required to maintain it in a consistent state. Likewise, the shorter the search path, the more efficient the lookup process.

Several structured overlay networks and routing algorithms were designed over the last decade, from *Chord* [SMK+01] that is based on an oriented ring, to *CAN (Content-Addressable Network)* [RFSH01] with its multi-dimensional *Cartesian* coordinate space, to *Pastry* [RD01a] which is based on the *Plaxton* [PRR97] structure, to *Tapestry* [ZKJ01], *Kademlia* [MM02], *Kelips* [GBL+03], *Viceroy* [MNR02] and many more [ZKW05, MBRI03], all with different trade-offs between routing complexity, maintenance overhead and memory footprint.

Although key-based routing algorithms are far more efficient than other previously described routing algorithms, the fact that they are based on collaboration implies several issues which are discussed next.

Structured overlay networks have long been considered to tolerate churn. However, subsequent studies [LSG+04] showed that well-known *DHT*s suffered from churn. Research [RGRK04] therefore explored the critieria impacting churn tolerance such as periodic versus reactive recovery, the choice of nearby versus distant neighbours *etc.*

Peer-to-peer networks have also been shown to implicitly suffer from attacks known as *Sybil* [Dou02] and *Eclipse* [SNDW06]. The *Sybil* attack consists of an attacker that generates enough virtual nodes to take over a large portion of the overlay network's identifier space. Therefore, a malicious node could, for instance, control all the replicas of an object. On the other hand, the *Eclipse* attack consists of malicious nodes corrupting honest nodes' routing table in order to increase the number of requests passing through such Byzantine nodes. Although these issues are very difficult to deal with, some routing algorithms were improved [CDG+02, DLLKA05, HKD07] to cope with such attacks.

Routing algorithms in peer-to-peer systems rely on the collaboration of the nodes populating the network. Since peer-to-peer networks are, by nature, untrustworthy, a single node being unwilling to cooperate *e.g.* to contribute to the routing process, to store the object it has been given the responsibility for *etc.* suffices to harm

the system and its users. *KBR* algorithms tend to rely on iterative routing instead of recursive routing to minimise the impact and more easily detect such malicious nodes, though such a design makes the lookup process less efficient.

Former peer-to-peer file sharing applications' problems with free riders came from the lack of incentive for the users to contribute their files and/or bandwitdh. In the last decade, research started exploring a completely different but more promising way to cope with such behaviours by enforcing collaboration in peer-to-peer networks. Systems bringing incentive to peer-to-peer systems fall in two categories. The first class is composed of systems relying on resource bidding. These systems [CN03, CGM02, MT03b, BLV05] guarantee that, for instance, whenever a node wants to store a block of data on another node, it must offer this node some local storage space in return. The second class is composed of reputation systems. Those systems [WV03, SS02, ZH07, DMS03, MT03a] dynamically keep track of nodes' behaviour in a completely decentralised way. Then, reputation is propagated through the system and correlation is made to detect Byzantine behaviours. Although both categories suffer some limitations, they represent the most promising solutions for enforcing collaboration in peer-to-peer networks.

The remainder of this section focuses on detailing two very different structured overlay networks along with their key-based routing algorithm, giving the reader a good understanding of the trade-offs involved in the design of such systems: *Chord* achieves high scalability while *Kelips* focuses on ensuring constant time lookups.

2.1.3.1 Chord

Chord is a *KBR* algorithm relying on a structured overlay network in which nodes are assigned random identifiers through the use of a hash function, for instance by applying *SHA (Secure Hash Algorithm)* on the node's *IP* address.

Identifiers are ordered in an *identifier circle* modulo 2^m. Key k is assigned to the node whose identifier is equal to or follows k in the identifier space. This node is called the *successor* of key k, denoted $successor(k)$. Note that the successor basically corresponds to the home or root node in other protocols *i.e.* the node responsible for the identifier.

The idea of *Chord* is to provide efficient routing *i.e.* to locate the successor of a given key, by relying on a very small amount of local information.

First, each node need only be aware of its successor node on the circle, ensuring that by passing the query around the circle, the key's successor will eventually be reached. Although *Chord* nodes do maintain a link with their successor and therefore ensure that all lookups can be resolved correctly, this routing scheme is very inefficient *i.e.* $O(\eta)$.

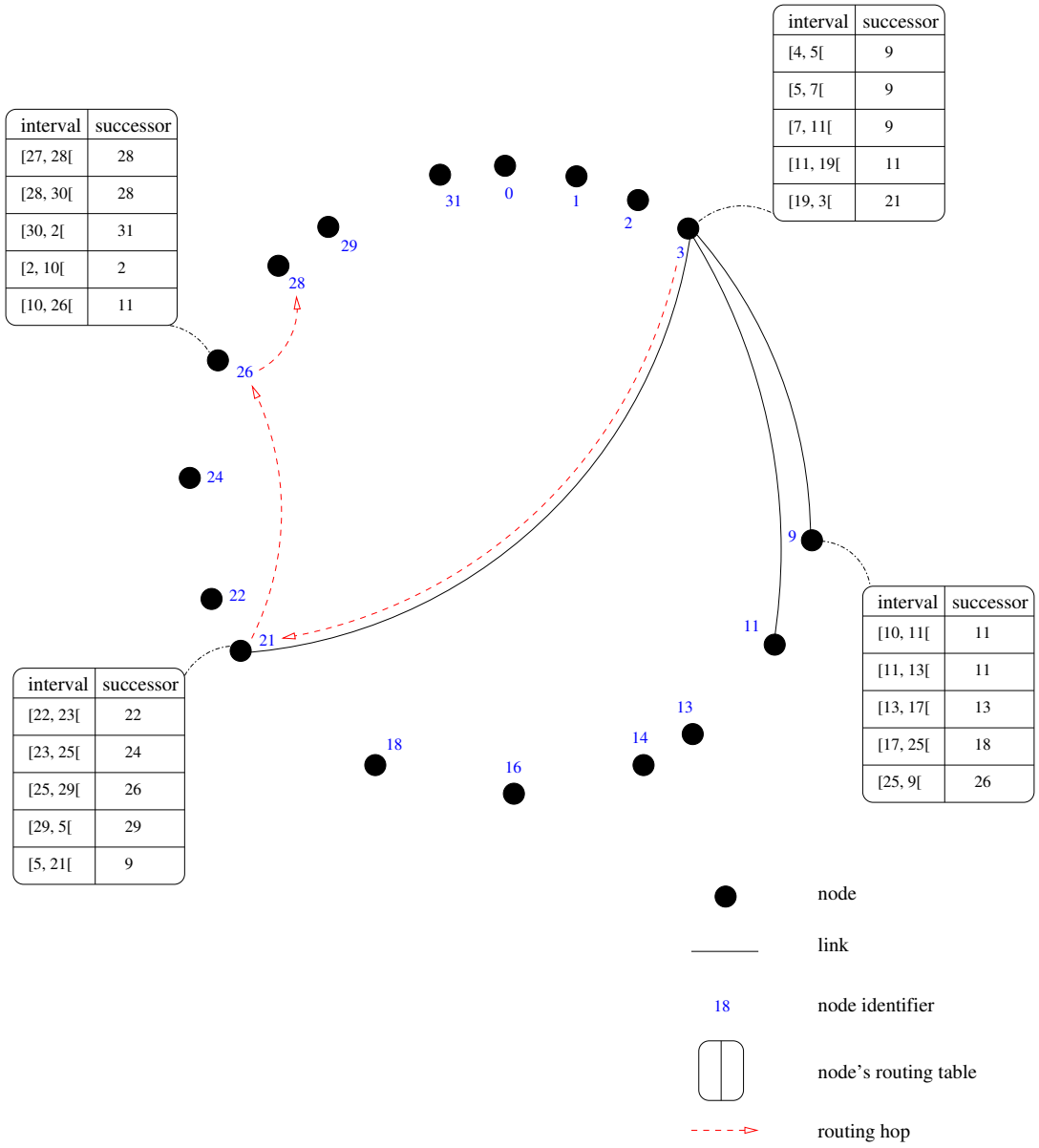


Figure 2.4: A *Chord* network of degree 5 with 17 nodes

To accelerate the process, *Chord* maintains additional, but few compared to the network size, routing links. Each node maintains a routing table, known as the *finger table*, composed of m entries. Recall that the maximum number of nodes in the network has been set to 2^m. Therefore, by keeping only m links, the *finger table* grows logarithmically with the size of the network. In the routing table of node n, the i^{th} entry contains the identifier of the first node, s, that succeeds n by at least 2^{i-1} on the identifier circle:

$$s = successor(n + 2^{i-1}) \ mod \ m, \qquad 1 < i < m$$

The system calls s the i^{th} *finger* of node n. A finger entry in *Chord* contains both

the identifier and *IP* address of the node. Note that the first entry—*i.e.* index zero—of the finger table points to what has been earlier called the node's successor.

This scheme has two important characteristics. First, each node stores information about only a small number of nodes, and knows more about close nodes than nodes on the other side of the circle. Second, often, a node's finger table does not contain enough information to perform the resolution by itself. Therefore, a node wishing to locate a node it does not know about would have no choice but to take a node in its finger table, whose identifier is closer to the key k than its own, and ask it to carry on the lookup process. By repeating this operation, every node without the necessary information forwards the request so that every step brings the request closer to the target node and eventually reaches it.

Figure 2.4 shows the organisation of a *Chord* ring along with the finger table of some nodes. In this illustrated network, node 3 issues a lookup on key 27 which is held by node 28. Since node 3 does not have the location of node 28 in its finger table, it forwards the request to the node 21, located in the farthest interval $[19, 3[$. Once node 21 receives the request, it inspects its finger table and notices that it cannot resolve the mapping either, hence forwards the message to the node 26 located in interval $[25, 29[$. One can easily notice that the interval is shrinking by half every time the request is forwarded. At this point, node 26 knows that node 28, located in interval $[27, 28[$, is responsible for the key 27 and therefore returns to the requesting node 3 the *IP* address of node 28, node 26's successor.

Chord provides a protocol for resolving an identifier into an *IP* address in a completely decentralised manner. Assuming the network is composed of η nodes, *Chord* resolves lookups in $O\big(log(\eta)\big)$ messages while nodes are required to maintain links to $O\big(log(\eta)\big)$ other nodes.

2.1.3.2 Kelips

As previously explained, malicious nodes involved in the routing process can interfere and harm the system by refusing to comply with the protocol. Since the longer the routing path, the higher the probability of a malicious node interfering, *Kelips* was designed to achieve $O\big(1\big)$ routing complexity at the cost of increased storage overhead. Considering a network of η nodes, *Kelips* uses $O\big(\sqrt{\eta}\big)$ space per node. This soft state suffices to resolve lookups with $O\big(1\big)$ time and message complexity at the cost of more background communication.

Kelips consists of κ virtual groups identified from 0 to $\kappa - 1$. Each node lies in a group determined by using a consistent hashing function such as *SHA-1*, applied on the node's *IP* address for instance. The distribution property of hash functions ensures that, with high probability, the number of nodes in each group will be close

to $\frac{\eta}{\kappa}$.

Nodes' soft state consists of two data structures. The first one, known as *Contacts*, contains the address of a small number of nodes lying in each of the other $\kappa - 1$ groups. The second data structure, known as *Neighbours*, contains the address of all the other nodes in the same group, hence the location of the home nodes of any key falling in this group.

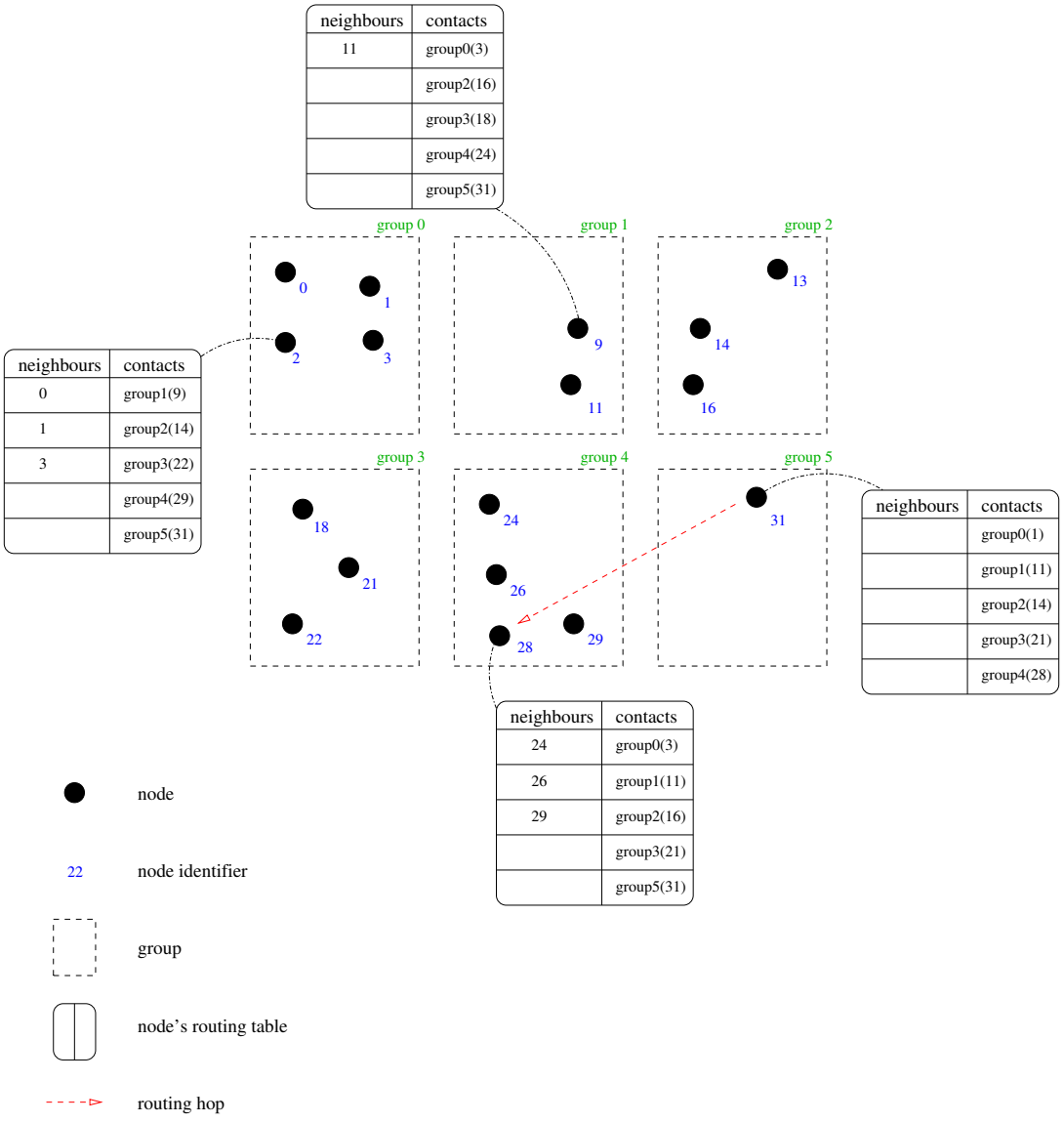


Figure 2.5: A *Kelips* network for 36 nodes

The routing algorithm consists of the following steps. The node wishing to locate the root node of a given key starts by extracting the group identifier corresponding to that key, for instance by using the m most significant bits of the key. The node looks into its *Contacts* table, and, if not located in its own group, picks a node belonging to the destination group. It then sends a message to this node. When the

node receives the message, it simply looks in its *Neighbours* data structure to locate the root node of the given key.

Figure 2.5 illustrates a *Kelips* network designed for $\eta = 36$ nodes. The network is composed of $\kappa = \sqrt{36} = 6$ groups while every group can contain up to 6 nodes. The *Contacts* and *Neighbours* data structures are detailed for some nodes. Finally, an example of a routing process is depicted. Node 31 wants to find the node responsible for the key 25. *Kelips* follows the same rule as *Chord*, the node whose identifier is equal to or follows the key is considered its root. Node 31 starts by extracting the group number corresponding to the key 25: group 4. It then picks in its *Contacts* a node lying in the group 4, node 28, and sends it a message request. When node 28 receives the request, its looks in its *Neighbours* data structure and notices that node 26 is the root node of key 25. Therefore, node 28 directly returns node 31 the address of root node 25.

Kelips ensures a $O(1)$ routing complexity because a single hop is required to locate the home node: either directly within the node's group or by contacting a node from the group in which lies the target node. Aside from the obvious performance benefits, this scheme allows the system to more easily detect malicious nodes since fewer intermediate nodes are asked to contribute to the routing process. However, *Kelips* does not scale well as the more nodes in the network, the more often the state changes, hence more communication is required to keep the state consistent.

2.1.4 Social Overlay Networks

A social network is a social structure made of individuals connected through relationships such as friendship, kindship, belief, knowledge, collaboration or just interest.

Such networks provide very interesting properties. Firstly, since routing in such networks consists in traversing nodes with some degree of trust, the routing process is less likely to be disturbed by malicious nodes than in other overlay networks. Secondly, many social networks exhibit the small-world phenomenon in which a generally short chain of acquaintances exists connecting one arbitrary node to any other node. Thus, the distance between two randomly chosen nodes grows proportionally to the logarithm of the number of nodes η in the network. Thirdly, in many applications, a node's acquaintances share the same interests such that most objects requested by that node will already be held by its neighbours, hence, greatly improving data retrieval.

Recently, research was conducted regarding the application of social behaviours to overlay networks in order to improve the performance and reliability of routing algorithms. Indeed, some existing networks, such as peer-to-peer file sharing communities, have been shown [IRF04] to exhibit small-world patterns, while non-small-world

networks have been improved [SMZ03] through the addition of social links.

As shown on *Figure 2.6*, nodes are connected to their friends, forming multiple loosely connected groups. In addition, every node could maintain a few links on a structured overlay in order to guarantee monotonic lookup progression. Indeed, if a node does not have any friend connection located closer to the target identifier, structured links can be used to move forward, hence guaranteeing liveness.

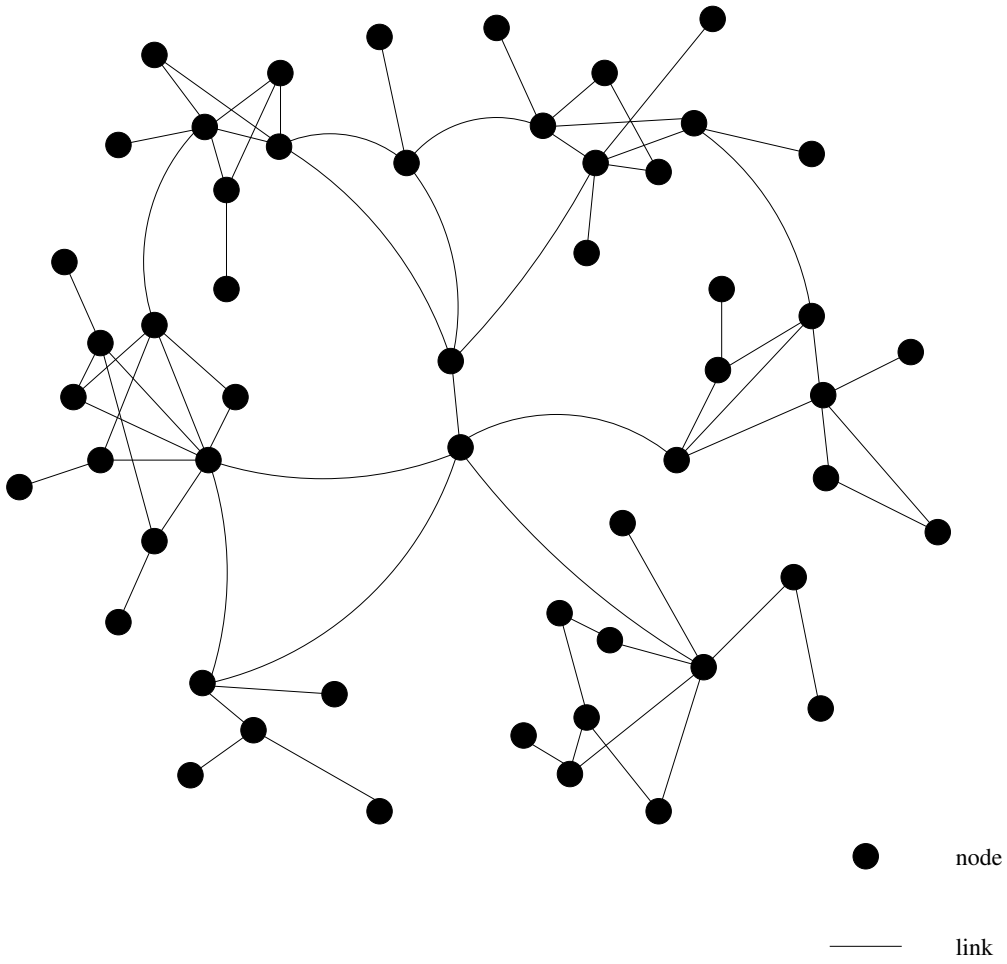


Figure 2.6: A small-world-based social overlay network

Some applications have been making use of social connections such as *Turtle* [PCT04], a peer-to-peer file sharing application relying on the friend relationship. *Turtle*'s routing protocol is similar to those of unstructured overlay networks, consisting of forwarding the request to the neighbour nodes, hence flooding the network. Other projects tried to enhance existing systems, such as *SPROUT* [MGGM04] which augments the *Chord* structured overlay network with social links in order to take advantage of the small-world network properties when possible and to rely on the structured nature of the underlying network otherwise. Finally, some social overlay networks were designed from the ground up to take advantage of the small-world

phenomenon. For instance, the *SWOP (Small World Overlay Protocol)* [And04] achieves improved object lookup performance over the existing routing protocols but also provides efficient replication especially regarding popular content.

2.2 Distributed Hash Tables

A *DHT (Distributed Hash Table)* provides a hash table abstraction on top of a peer-to-peer overlay network. Such a service aggregates the network peers' storage resources providing a distributed data structure. A *DHT* provides a way to store a block of data β given an address—*a.k.a.* storage key—α, usually through an interface [DZD+03] as simple as `Put`(α, β) and `Get`(α).

In order for the service to be efficient but also scalable, *DHT*s make use of key-based routing algorithms. For instance, the distributed hash table *PAST* [RD01b] is built upon the *Pastry KBR* while *DHash* is based on the *Chord* overlay network.

As discussed through the remainder of this section, redundancy is an absolute requirement for ensuring availability, durability and integrity. Distributed hash tables therefore abstract the process of replicating data and maintaining replication consistency [KWR06] as nodes fail and join the system.

Indeed, considering a *DHT* in which every block is stored by a single node, availability could not be ensured since the failure of this node would make all the blocks it was responsible for storing inaccessible. Besides, assuming that the node crashes permanently, the block would be lost forever. Redundancy is therefore an absolute requirement for ensuring both availability and durability.

Furthermore, in a system lacking redundancy, nothing would prevent the home node from altering the data content and/or returning fake content to a client's request. Although systems such as *SUNDR* [LKMS04] ensure integrity without relying on trusted storage servers, clients cannot retrieve the block's latest valid content if the block's only storage node does not want to cooperate and keeps acting maliciously. In order to provide the clients the assurance of valid data retrieval, the system must rely on redundancy so that a block is always stored by a set of nodes.

There are basically two ways of achieving redundancy, either through replication or network coding schemes. Replication [SS05, JGH+98] consists of storing multiple identical instances of an object on different nodes, hence increasing availability and durability. Network coding schemes [OSV09] however rely on error-redundant codes such as *Reed-Solomon*, an erasure code [DGWR07] widely used in *DVD (Digital Versatile Disc)*. Instead of plain object replication, erasure code schemes divide the object into m fragments and recode them into n segments, where $n > m$. The n segments are then stored in the *DHT*. The rate of encoding $r = \frac{m}{n}$ increases the

storage cost by a factor of $\frac{1}{r}$. The key property of erasure codes is that the original object can be reconstructed from any m segments. For example, using a ratio $r = \frac{1}{4}$, a block is divided into $m = 16$ fragments and encoded into $n = 64$ segments, increasing the storage cost by a factor of four. Then, a client able to retrieve sixteen segments out of the sixty four present in the system would be able to reconstruct the original object. Noteworthy is that replication represents a subset of erasure codes where the number of segments n is one *i.e.* a single segment is enough to reconstitute the original object.

Network coding schemes are very interesting because they require less storage space in order to achieve the same degree of availability and durability as through standard replication. As an example, assuming that ten percent of the ten million machines populating a network are down, replication ensures 99% availability by storing two replicas of each block. However, erasure codes can achieve over 99.9999998% yet consume the same amount of storage and bandwidth than their replication counterpart.

Unfortunately, network coding schemes all suffer from the same problem. As nodes fail, the system loses the segments belonging to the network-coded objects the nodes were storing. In order to avoid losing them all, the system must, for every object involved, periodically refresh the missing segments. This refreshing process consists of reconstructing the object, re-computing all the segments and then re-storing the missing segments on other storage nodes. Unfortunately, this process is extremely costly, especially for large objects, though network coding schemes were designed to rarely require refreshing segments.

Figure 2.7 illustrates *DHash*, a *Chord*-based distributed hash table in which blocks are replicated on the nodes following the home node, known as the *neighbour*s. Since nodes with close identifiers are, with high probability, located in very different geographic places, storing replicas on such nodes ensures a low rate of correlated failures. Note that, whenever the home node fails, the *Chord* protocol takes over and assigns a new home node to the orphan objects. In addition, the *DHT* makes sure the replication ratio is maintained at all times by generating additional replicas if required.

Although network coding has been studied [WK02] for decades and applied in numerous research systems as well as commercial products [Wua], replication [SS05] remains the most widely used technique to provide redundancy in peer-to-peer networks.

Given that redundancy is required for ensuring availability and durability, the system must guarantee consistency among the replicas. Unfortunately, like every *Internet*-based distributed system, *DHT*s are built on top of an asynchronous physical network, making it impossible [FLP85] to distinguish slow from faulty nodes. This

property impacts the consistency algorithms which must often take further steps should the nodes not be responding or acting maliciously.

*DHT*s rely on consensus algorithms in order to cope with Byzantine behaviours in asynchronous networks. As summarised by *Chockler et al.* [CGKV08], such algorithms vary in several dimensions from the consistency guarantees, to the number of failures tolerated, to the performance achieved. Consensus algorithms can be classified in two categories: agreement and quorum protocols.

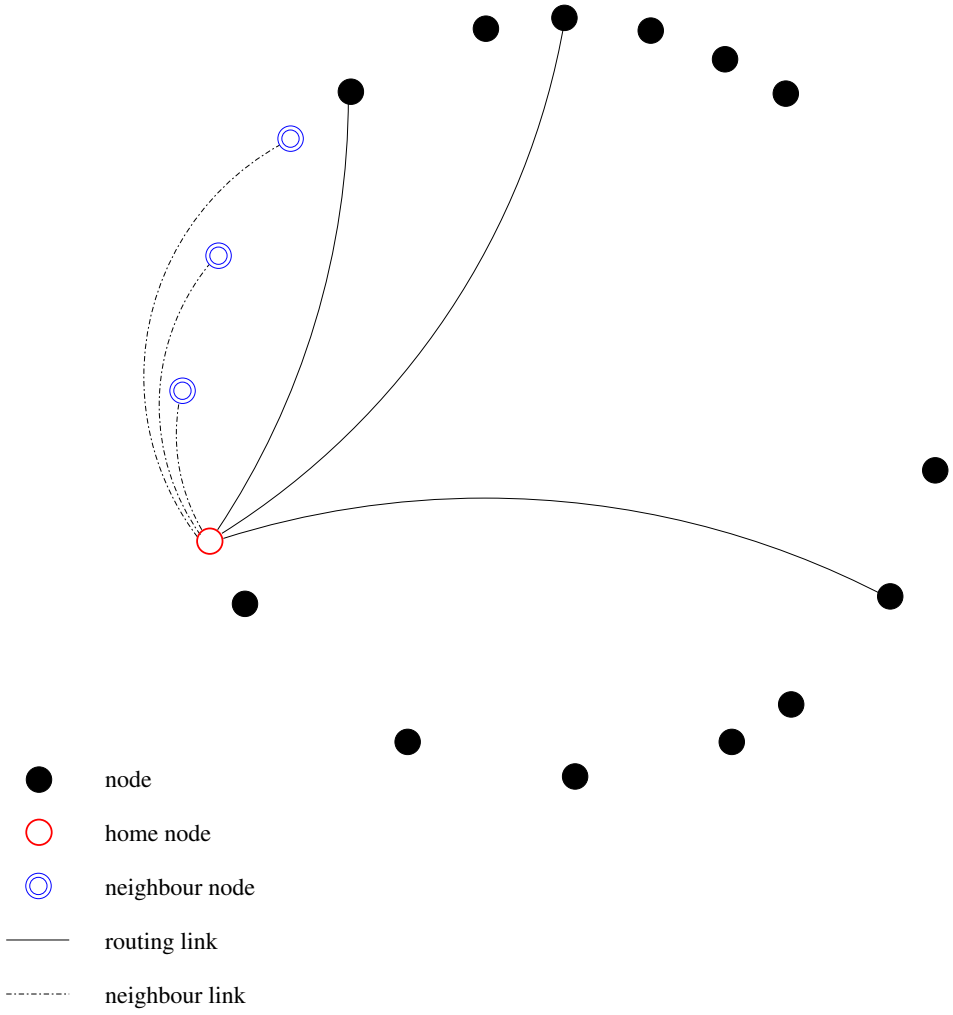


Figure 2.7: The replication-based *DHash* distributed hash table

Byzantine agreement protocols such as the *BFT (Byzantine Fault Tolerant)* [CL99] protocol and *Paxos* [Lam98, Lam01] achieve consensus through voting and can tolerate up to γ Byzantine nodes by relying on $\varphi \geq 3\gamma + 1$ servers. Such algorithms work as follows. A client willing to perform an operation starts by sending a request to a server *i.e.* the leader. The server having received the client's request then forwards it to the other servers. Every server receiving such a vote request responds to every other server, hence leading to a consensus. Finally, the leader

server transmits the servers' decision back to the client. This multi-phase protocol is illustrated by *Figure 2.8*. Although such algorithms are extremely powerful for dealing with Byzantine behaviours, they unfortunately suffer from the several rounds of communication which generate a number of messages quadratically proportional to the number of servers involved. Agreement algorithms are thus often considered as being too expensive [DW01, Bus07] for many applications. Noteworthy is that many improvements and optimisations have been developed over the years. For instance, *Borran et al.* [BS10] proposed a leader-free Byzantine consensus algorithm while others [CSP07, LMZ09] presented optimisations of the *Paxos* algorithm regarding specific configurations: multiple coordinators, reduced number of rounds in the absence of failures *etc.*

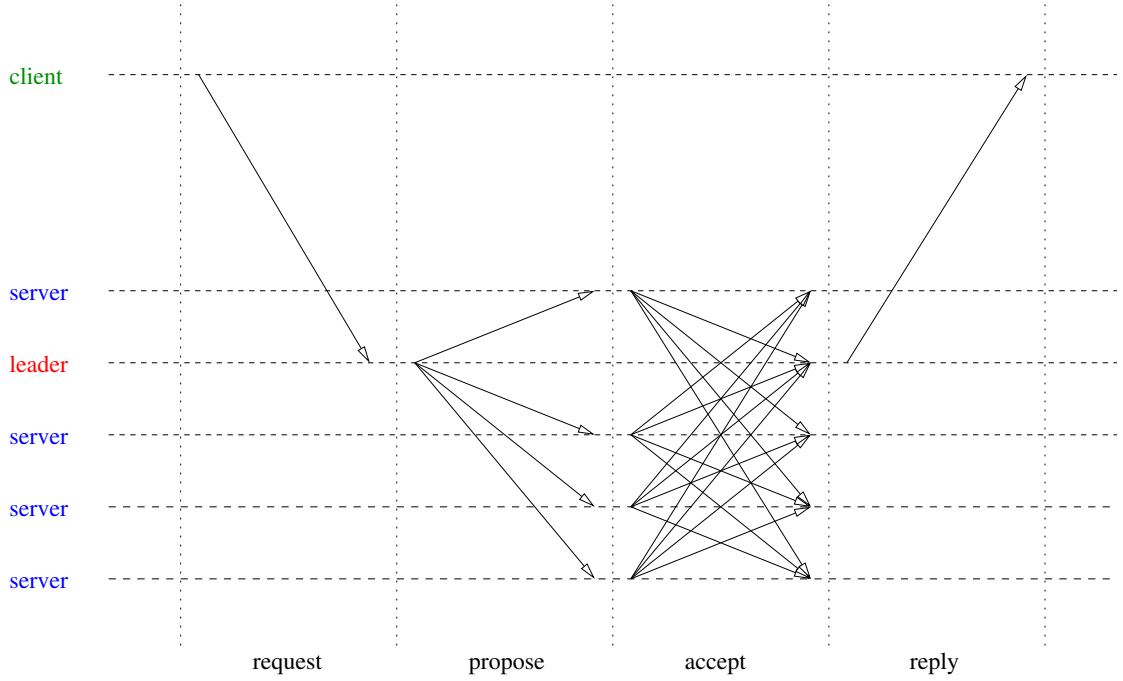


Figure 2.8: The *Paxos* agreement protocol

On the other hand, quorum-based algorithms [AJ92, MR97, GKLQ07, MAD02] consist of retrieving a subset of the replicas to make sure to identify the latest valid version of the object. Quorums rely on the property of intersection in order to minimise the number of storage nodes to contact but also to prevent conflicts. As for agreement protocols, quorum-based algorithms have been the subject of numerous research projects which have led to further improvements, especially in the field power management and mobile networks [BF08, GDZ+05, KLW11].

Although many quorum-based algorithms have been presented throughout the history of distributed computing, *Gifford et al.*'s [Gif79] quorum-based protocol is detailed next because of its simplicity.

Considering a distributed system in which blocks are replicated on φ storage nodes, a client willing to perform an operation must acquire a quorum complying with the following rules, where ζ_r and ζ_w represent read and write quorums' cardinality, respectively:

1. $\zeta_r + \zeta_w > \varphi$
2. $\zeta_w > \frac{\varphi}{2}$

The first rule prevents read-write conflicts while the second rule prevents write-write conflicts, both contributing to maintain serialisability.

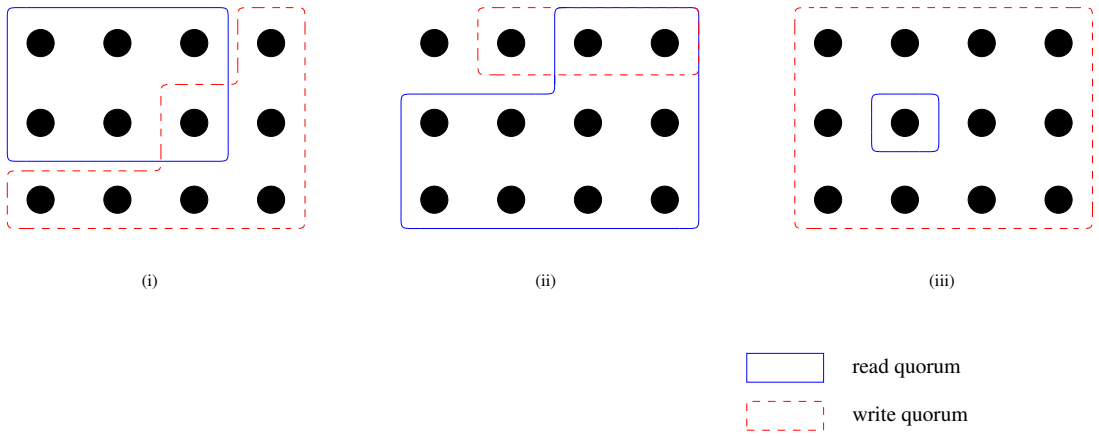


Figure 2.9: Three *Gifford* quorum configurations

Figure 2.9 depicts a set of twelve storage nodes in three different quorum configurations. In the first configuration, read and write quorums are composed of six and seven nodes, respectively, such that the quorums intersect on a single replica. Therefore, assuming that a client updates an object by contacting seven replicas out of the twelve present in the system, a subsequent read through a quorum of six replicas will inevitably provide the client the recently updated version. The second configuration does not comply with the *Gifford* quorum rules since $\zeta_w > \frac{\varphi}{2}$ is not respected. Therefore, up to four clients could modify the object concurrently, leading the system to an inconsistent state. Finally, the last configuration is generally referred to as *ROWA (Read One, Write All)*. Indeed, in such a configuration, reading an object requires the client to contact a single node while all the nodes must respond positively whenever an object is updated.

It is however extremely important to note that most of the configurations depicted in *Figure 2.9* would not be suitable for Byzantine environments. Indeed, in case of arbitrary failures, such as in peer-to-peer networks, $\varphi \geq 3\gamma + 1$ storage nodes are

required to tolerate up to γ Byzantine nodes while read and write quorums must contain, at least, $2\gamma + 1$ replicas.

To conclude, agreement protocols provide more expressivity than quorum protocols. Indeed, while agreement protocols can achieve consensus on virtually any kind of operation, quorum protocols are limited to reads and writes. However, quorum algorithms have proved to be well suited for peer-to-peer file systems which are built on top of distributed hash tables, such constructs providing functionalities as basic as `Put`(α, β) and `Get`(α). Besides, both agreement and quorum protocols are equally constrainable from the client's perspective since at most $2\gamma + 1$ instances of the block must be retrieved in order to cope with Byzantine behaviours. On the other hand, agreement protocols' several voting phases imply a high number of message exchanges. Thus, agreement protocols are often considered as being very expensive [BBB+04, DW01, Bus07], especially in the context of peer-to-peer file systems though many projects have been making use of those [KBC+00, ABC+02].

2.3 Peer-to-Peer File Systems

The very first distributed systems targeted local-area networks, the most famous and still widely used being *NFS (Network File System)* [Osa88]. Such local-area distributed file systems are characterised by a low network latency as well as trustworthy clients and servers, both evolving within a single administrative domain.

Unlike local-area network file systems, *AFS (Andrew File System)* [HKM+88a] addresses larger networks characterised by higher latencies and a larger number of computers. Such file systems rely on loose caching policies [SS96] in order to reduce the communication between the clients and the servers. Moreover, systems such as *Coda* [SKK+90] and *Ficus* [JGH+98] enable offline access through the use of optimistic replication [SS05], applying modifications once the computer re-connects to the network.

Many other file systems were designed for small- and medium-sized networks, all with different objectives and constraints including *Kosha* [BJZH04] which equips *NFS* with redundancy over a scalable network, *xFS* [WA93, ADN+95], a wide-area file system relying on massive caching techniques, *Plan9* [PPD+95], a distributed computing environment following the *UNIX* philosophy and *LBFS (Low Bandwidth File System)* [MCM01] which reduces communications by relying on indexes and applying *Rabin* fingerprints to the chunks of data.

Unfortunately, these distributed file systems rely on trusted and often centralised servers making them impractical in more open environments. *SFS (Self-Certifying File System)* [MKKW99, KSMK03, Maz01, FKM02] addresses this issue by relying

on multiple self-certifying domains rather than a single global open network. Since the domains are independent, the domain management is assured by the local authority with its own rules and policies. The self-certifying property of *SFS* makes it impossible for an attacker to pretend to be or belong to another domain. *SFS* therefore achieves scalability through openness although the domains' independence implies that the failure of a single domain renders all the users, groups, files and directories of this domain unavailable.

Peer-to-peer networks have been shown [BDET00] to exhibit very interesting properties for building highly available and reliable file systems. The remainder of this section discusses in detail some of those peer-to-peer file systems especially regarding their capacity to scale, cope with Byzantine behaviours but also provide common file system features such as access control.

2.3.1 Pangaea

Pangaea [SKKM02] is a wide-area read/write file system which relies on an *ad hoc* decentralised storage infrastructure of trusted servers. *Pangaea* aims at providing clients efficient data access through the use of pervasive replication.

In order to optimise the data placement, the nodes of the system are split into disjoint *region*s. A region is composed of nodes grouped according to their network latency. Every node maintains a global state of the whole system including the list of the nodes of the region, their network latency and free disk capacity, the location of the root directory's replicas, the list of the regions *etc.* This information is propagated throughout the network periodically by means of an epidemic protocol.

Pangaea maintains, for every file, a distributed and highly connected graph of the nodes storing replicas known as *gold replica*s. Such replicas are statically defined at file creation and are used to maintain a minimum replication ratio at all times. In addition, *bronze replica*s are also connected to the graph in a loose manner. Indeed, *bronze replica*s are created in a dynamic way *i.e.* every time a node accesses the file. This replication graph, composed of both *gold replica*s and *bronze replica*s, is used to propagate updates throughout the network in an efficient way.

Figure 2.10 illustrates two directories along with both their gold and bronze replicas. Every directory replica instance contains the locations of the entries, being files or sub-directories. These locations are represented on the figure by the *reference*s. In addition, although not depicted by the figure, every replica maintains a *backpointer* to the parent directory which is used to update the parent directory should the gold replicas of a sub-entry be moved to another node.

The data structure described above has the advantage of distinguishing the replication of the directory from the replication of the objects, files and directories it

contains. Therefore, adding or removing bronze replicas only requires updating the graph related to this object while leaving both the parent directory and the potential sub-objects out of the process. Indeed, the only operation requiring updating of the parent directory is the modification of a gold replica's location.

The modifications applied to a replica are propagated throughout the graph following the edges connecting the replication nodes. Note that an operation description—*e.g.* `create file 'bar' in directory '/foo/'`—rather than the new object's state is propagated to the replication nodes. *Pangaea* makes use of the *last-write-wins* consistency model by relying on global timestamps through the use of a *NTP (Network Time Protocol)* server. However, directory conflicts are resolved automatically if possible or left to the user otherwise.

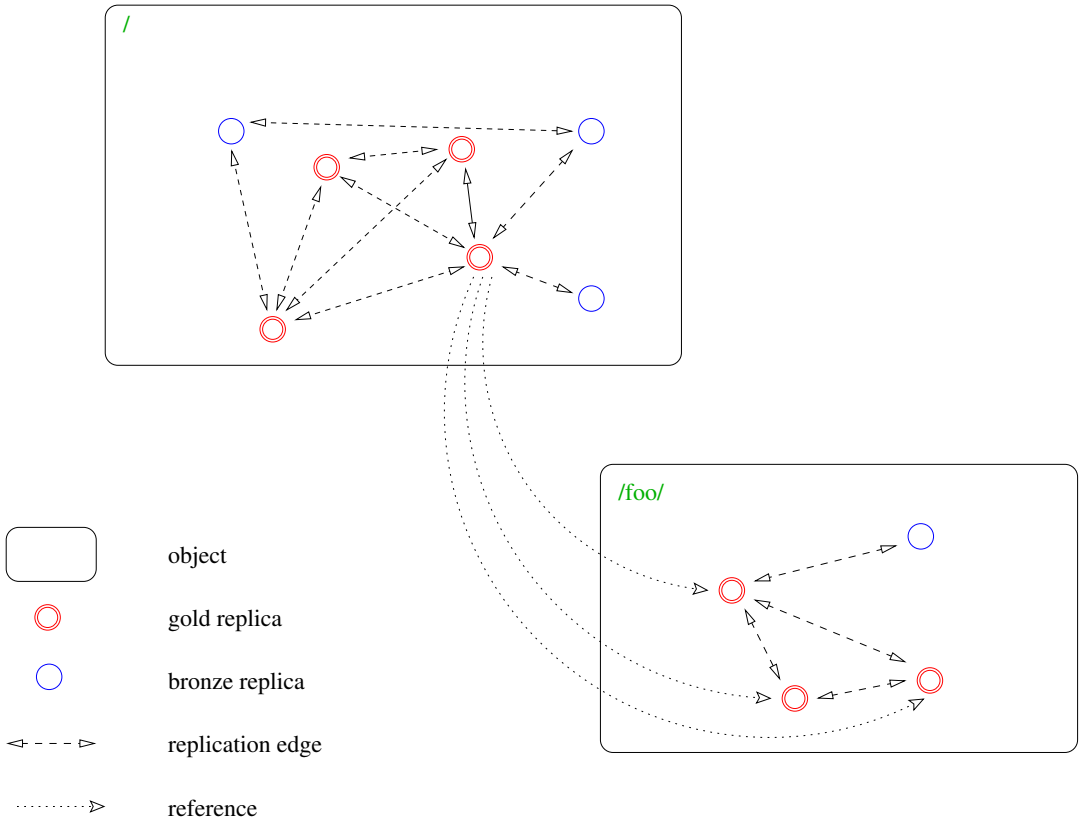


Figure 2.10: *Pangaea* file system representation

Although *Pangaea* provides a powerful storage infrastructure through localised replication, the assumption of a trustworthy network makes it impractical for most environments. Indeed, since both authentication and access control is handled by the trusted servers, a single Byzantine node could easily harm the whole system.

2.3.2 OceanStore

OceanStore [KBC+00] is a generic distributed storage infrastructure relying on the *Tapestry* [ZKJ01] *DOLR (Decentralised Object Location and Routing)*. *OceanStore* aims at providing a wide range of consistency models in order for applications to balance the trade-offs between performance and consistency. *OceanStore*'s architecture is partitioned into two levels or *tier*s.

The first level, known as the *primary tier*, is composed of highly available nodes divided into multiple groups. Each group is responsible for a subset of the objects of the system. The nodes belonging to a group store the *primary replica*s of the objects the group is in charge of. The *BFT* [CL99] agreement algorithm is used by the group members for authorising, validating and applying operations on the replicas despite the potential presence of Byzantine nodes. Note however that such an algorithm is expensive as it requires three communication rounds between the servers to perform a single operation, generating $O(\psi^2)$ messages, assuming every group is composed of ψ servers. Therefore, the nodes composing the primary tier must be very powerful, well connected and highly available to handle the high network load.

The *secondary tier* is composed of more transient nodes with high latency and low bandwidth such as personal computers for instance. This level constitutes the mass storage capacity of the system in which *secondary replica*s are created in order to improve local accesses.

As *Figure 2.11* illustrates, whenever a client node modifies an object, being a file or a directory, a request is sent to the primary replication nodes as well as some randomly chosen secondary replication nodes. While primary servers serialise and verify the operation validity by running the *BFT* algorithm, the request is propagated to the other secondary replication nodes in an epidemic way. Finally, once approved, a confirmation is propagated throughout the network, sealing all the secondary replicas.

A certificate is attached to every object's version, asserting the approval of the primary tier. Since primary servers may be malicious, the certificate cannot be generated by a single server. Therefore, *OceanStore* relies on threshold signature schemes [AMN01] so that a certificate is considered valid if composed of $\lfloor \frac{\psi-1}{3} \rfloor + 1$ legitimate partial signatures.

OceanStore makes use of optimistic concurrency control for optimising operations' response times. In order to detect conflicts, the system implements a semantic detection mechanism based on predicates. Therefore, depending on the type of object, whenever a conflict is detected, a pre-defined list of operations is applied to the object such as inserting, replacing or truncating the data. Note however that, although this mechanism is generic enough to automatically resolve conflicts, the

system often lacks semantic information since objects are usually encrypted.

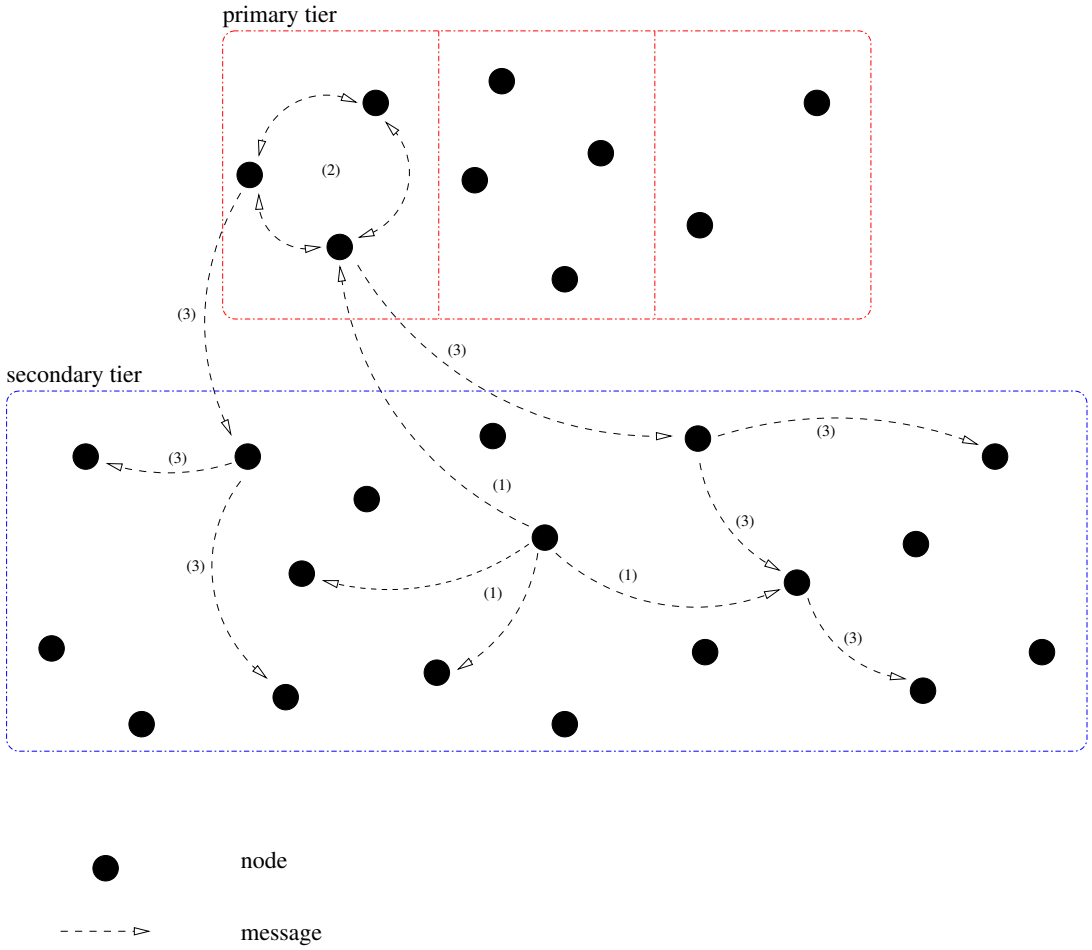


Figure 2.11: *OceanStore*'s organisation

Security is provided through the use of *ACL (Access Control List)*s. An *ACL* contains the public keys of the users allowed to access and modify the object. Whenever an object is modified, it is signed by the user and submitted to the primary replication servers. Then, every client and server retrieving the object can check the object's validity by verifying the digital signature.

Although *OceanStore* provides a powerful and flexible storage infrastructure, its hybrid architecture makes it more difficult to scale to large networks than other, completely decentralised, peer-to-peer file systems.

2.3.3 FARSITE

FARSITE (Federated, Available, and Reliable Storage for an Incompletely Trusted Environment) [ABC+02, DAB+02] is a file system based on an *ad hoc* partially decentralised storage infrastructure. *FARSITE* aims at emulating the behaviour

of a centralised file system such as *NTFS (NT File System)* in a medium-scale environment and without introducing new semantics such as file versions, conflict resolutions *etc.*

FARSITE has been designed to be deployed on the commodity hardware of medium-sized networks such as universities or companies. Such an environment is characterised by a high bandwidth network and transient nodes. Every node in *FARSITE* can play up to three roles: *client*s issue requests on behalf of end-users, *server*s store object replicas and *manager*s, as members of a *management group*, take part in administrating the system's metadata. Every management group is in charge of a subset of the file system's namespace. The members of a management group act collectively through the use of the *BFT* agreement protocol.

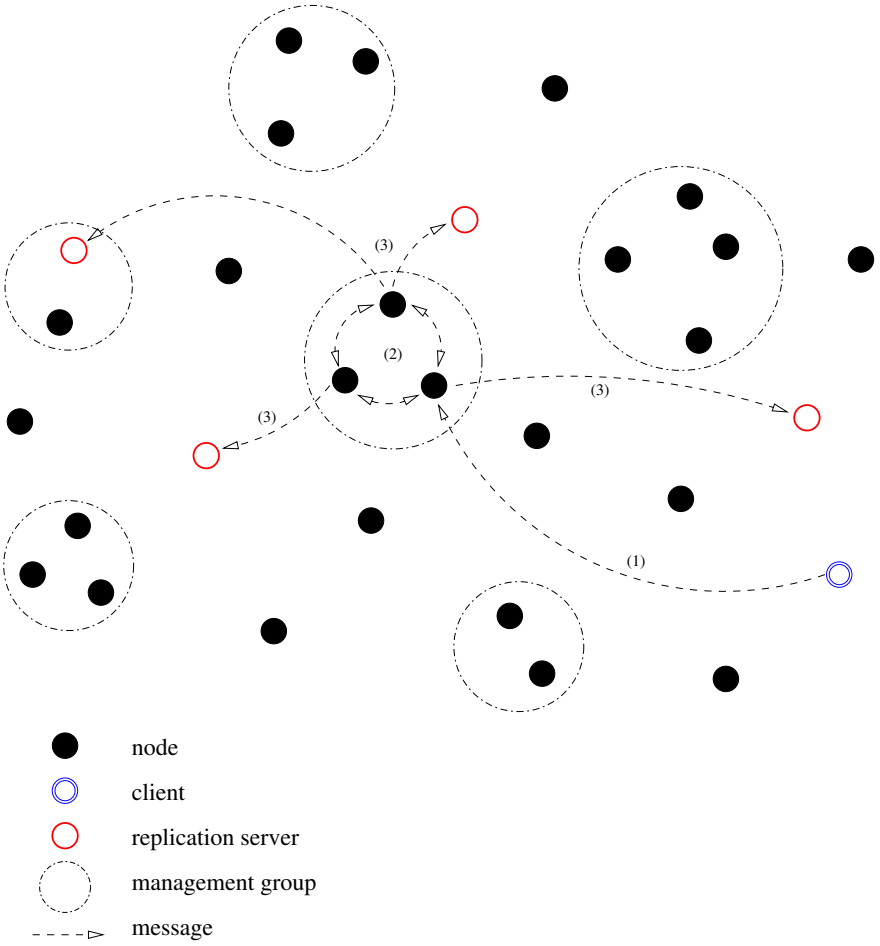


Figure 2.12: The *FARSITE* architecture

Every member of the management group maintains a copy of the metadata related to the namespace's subset it is in charge of. Access and modification requests are sent to the management group which runs a vote to serialise the operation. The management group keeps the locations of the replicas of every file lying in its namespace

as well as the hash of the file's content. A client willing to access a file starts by contacting the management group in order to locate the file's replicas. The client retrieves an instance of the file from one of these locations and checks the file's integrity by re-computing the hash of the content. The client then caches the file locally for subsequent accesses and/or modifications. Whenever modified, a hash of the new file's content is computed and sent to the management group which verifies that the user has the proper credentials for that operation. If the request is accepted, the servers storing the file's replicas are told to retrieve the new file directly from the client. *Figure 2.12* illustrates the described update protocol.

FARSITE ensures strong consistency guarantees by relying on leases. Whenever a client reads a file, it is granted a lease by the management group, guaranteeing the freshness of the client's local copy. *FARSITE* provides two types of lease: read leases ensure that the file will not be modified until the lease expires or is revoked while write leases guarantee an exclusive access to the file. Note that considering a client requesting a file, the system would immediately revoke the leases so that the eventual modifications are pushed back to the replication nodes, bringing the system back to a consistent state. Then, the requesting client could carry on its operation and retrieve the file in its latest form.

FARSITE provides security through the use of *convergent encryption*. Every file is assigned an *ACL* containing the public key of the users authorised to modify the object. Whenever a client requests an operation to the management group, a secure communication channel is established in order to authenticate the user. The system guarantees that unauthorised users cannot access a file through the following protocol. For every new file, the client generates a random symmetric key and encrypts the file's content with it. Then, the symmetric key is encrypted with the public key of every user having been granted permission to read the file. These encrypted symmetric keys are finally sent to the management group so that whenever a client requests a read operation, the management group returns the client its encrypted symmetric key. Then, the client can decrypt the symmetric key using the user's private key.

2.3.4 CFS

CFS (Chord File System) [DKK+01] is a completely decentralised file system relying on the *DHash* distributed hash table as a block storage abstraction. *CFS* aims at ensuring data integrity while balancing the storage load across the system's nodes. The particularity of this system lies in the fact that a single user can update it, such that *CFS* is often considered a read-only file system.

The entire *CFS* architecture relies on a block unit known as the *CHB (Content*

Hash Block). The special property of *CHB*s is that such blocks are self-certified, making the integrity verification process straightforward. Indeed, a *CHB*'s address is computed by applying a one-way function, such as the *SHA-1* hash function for instance, to the block's content. That way, whenever a client retrieves such a block, its integrity can be verified by re-computing the block's hash and checking if the fingerprint corresponds to the requested address.

Unlike other file system objects however, the root directory is stored in a *PKB (Public Key Block)*. *PKB*s are associated with a cryptographic key pair such that the block's address is computed by applying a one-way function to the public key. In addition, a digital signature of the block's content is embedded in the block for authenticity and integrity purposes. Noteworthy is that, unlike *CHB*s which are immutable, *PKB*s can be modified since their public key does not change over time. *PKB*s also embed a version number, which is increased whenever the object is updated, to differentiate the multiple instances of an object.

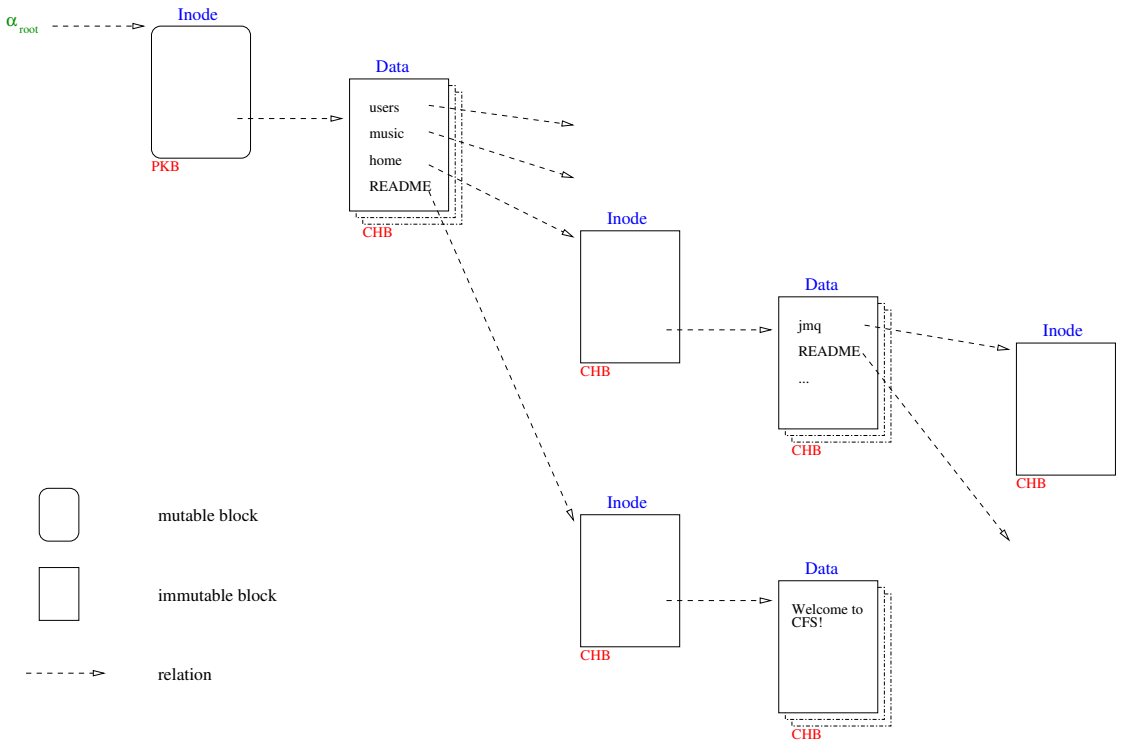


Figure 2.13: The *CFS* hierarchical organisation

Since the root directory block is mutable, the administrator can update the file system by re-signing the *PKB* with the private key he possesses. However, modifying the file `/home/README`, for instance, would imply, creating a new *CHB* for the file. Since the directory containing the file is a *CHB* as well, modifying a file also changes the directory content. Indeed, the file has a new content hence a new address and the directory content contains tuples of the form $(name, address)$. Since the address

for the file `/home/README` has changed, the directory must be updated accordingly. Finally, since the directory `/home/` just changed *i.e.* has a new address, the parent directory must be updated as well, and so on up to the root directory block. Therefore, modifying a single byte in a file implies updating the file system hierarchy up to the root block, which is in turn, re-signed by the administrator. Such a modification process is extremely expensive and inconceivable in a production environment. *Figure 2.13* illustrates the *CFS* hierarchical organisation based on the *UFS (UNIX File System)* in which metadata are stored in objects known as *inode*s.

The *CFS*'s approach regarding file representation differs from many systems such as *PAST* [RD01b]. In *PAST*, every file constitutes a *DHT* block while *CFS* split files into chunks of regular size. Splitting files into chunks has the advantage of better balancing the storage load across nodes. However, since files are composed of multiple data blocks, the *DHT* routing process is requested more often, potentially leading to performance loss and increased security threats.

2.3.5 Ivy

Ivy [MMGC02] is a multi-reader/multi-writer file system relying, like *CFS*, on the *DHash* distributed hash table. *Ivy*'s architecture is based on per-participant logs describing the modifications the given participant has performed on the file system. *Ivy* implements a log in the form of a chain of *records*. Every record is stored in a *CHB* while the *head* of the chain is referenced in a *PKB*, modifiable by the participant.

A *view* of the file system is composed of a set of such logs, as illustrated in *Figure 2.14*. The address of the log *head* blocks of the participants involved in the *view* are referenced in a *view* block. The *view* is stored in a *CHB*. The address of the *view* block identifies the file system and is therefore distributed to the users, enabling them to access and potentially modify it. Noteworthy is that since the *view* block is immutable, adding or removing a user to the *view* implies creating a new *view*.

The records are sequentially numbered while the highest attributed number n is kept in the log *head*. Every record is also identified by a vector timestamp $[n_i]$ corresponding to the highest sequence number n of the various i logs composing the *view* at the time of the record creation.

A user willing to modify the file system starts by reading the *head* block of every log composing the *view* in order to determine the vector timestamp to use for the new record. Then, the user adds a record to her log describing the operation performed such as the file path, offset, length and data for a `write` operation for instance.

Consulting the file system however requires the user to explore all the *view*'s logs from head to tail, looking for records related to the object and area of interest.

During this process, the client sorts the records according to their vector timestamps. For example, reading a file would require the client to locate all the `write` records related to the given file and intersecting with the area to read.

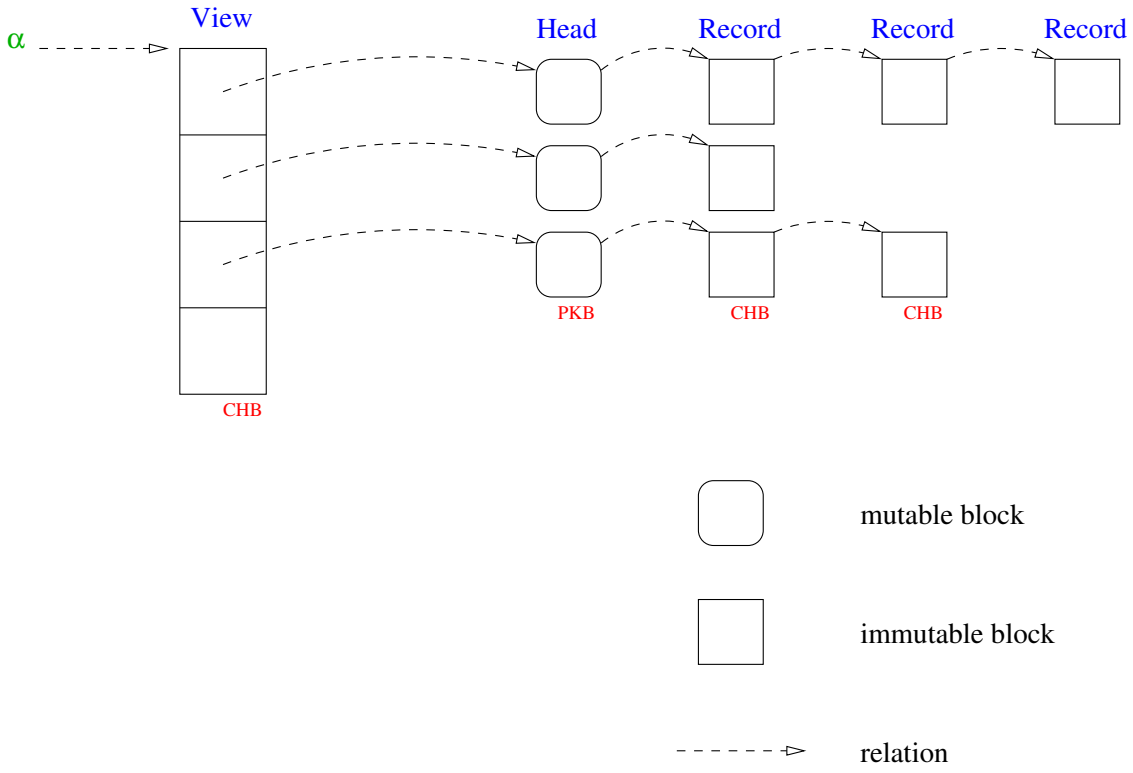


Figure 2.14: The *Ivy* log-based representation

Although this process of serialisation is extremely expensive, as the performance depends on the number of writers involved, representing a file system by means of per-participant logs has two advantages. Firstly, assuming that a malicious user is detected, a participant can easily use another *view* which does not include the malicious user. Indeed, any subsequent operation would simply make use of the new set of logs. Secondly, in case of a partitioned network, every participant can continue accessing the file system, assuming that every partition contains at least one copy of the logs. Although modifying the file system in such a partitioned environment can potentially lead to more conflicts, *Ivy* already provides the necessary tools for dealing with such situations.

Ivy makes use of optimistic concurrency control since operations are independently transcribed into records and serialised whenever a client needs to reconstitute the object. Note however that relying on independent per-participant logs does not prevent conflicts. Indeed, a scenario involving two participants modifying the same area of the same file based on the same file system state—*i.e.* vector timestamps—would obviously lead to a conflict. *Ivy* resolves such conflicts by ordering modifications

according to the user's public key, hence guaranteeing the same consistent view for all the participants.

Regarding security, *Ivy* does not provide access control mechanisms. Indeed, a user willing to restrict access to one of her files would have no choice but to manually encrypt it and distribute the encryption key to the authorised users making *Ivy* impractical for deployed environments.

2.3.6 Plutus

Plutus [KRS+03] is a decentralised file system built upon an *ad hoc* overlay network. *Plutus* aims at detecting and preventing unauthorised accesses, differentiating between read and write permissions and enabling the change of access rights.

Plutus access control is based on two ideas. Firstly, the key distribution process is delegated to the client, leading to better server scalability while allowing the user to set arbitrary policies. Secondly, in order to reduce the number of keys users must keep, files are grouped into *filegroup*s.

The aggregation mechanism of *filegroup*s prevents the number of keys the user has to manage to grow proportional to the number of files. An *RSA (Rivest Shamir Adleman)* key pair is associated with each *filegroup*. Files are grouped in *filegroup*s according to their sharing attributes so that two files shared by the same users will have the same encryption key. Since users tend to use the same access control rules for their files, the number of *filegroup*s a user's files belong to can be expected to be very low.

On the downside, using the same key for encrypting multiple files has the disadvantage that the same key encrypts more data, potentially increasing the vulnerability to known plaintext and ciphertext attacks. *Plutus* therefore uses unique encryption keys for different files and stores those keys in a *file-lockbox* whose key is then distributed to the users of the same *filegroup*.

Figure 2.15 illustrates the different keys involved in *Plutus*. Every file is split into data blocks, each of those blocks being encrypted with a unique symmetric *file-block* key. The *lockbox* contains all the *file-block* keys of the file and is encrypted with a symmetric *file-lockbox* key which is distributed to both readers and writers alike. Note that *file-lockbox* keys are the same for all the files belonging to the same *filegroup*. A hash of the file contents is computed for integrity purposes and signed with a *file-sign* private *RSA* key. The signature can subsequently be verified with a *file-verify* key *i.e.* the associated *RSA* public key. The *file-sign* key is handed to writers while the *file-verify* key is handed to readers so that the system can differentiate read from write access control. Thus, whenever a user modifies a file, she re-computes the hash and re-signs it. Readers however check the file's integrity

by verifying the signature with the *file-verify* key and then make sure the hashes are valid according to the file's contents.

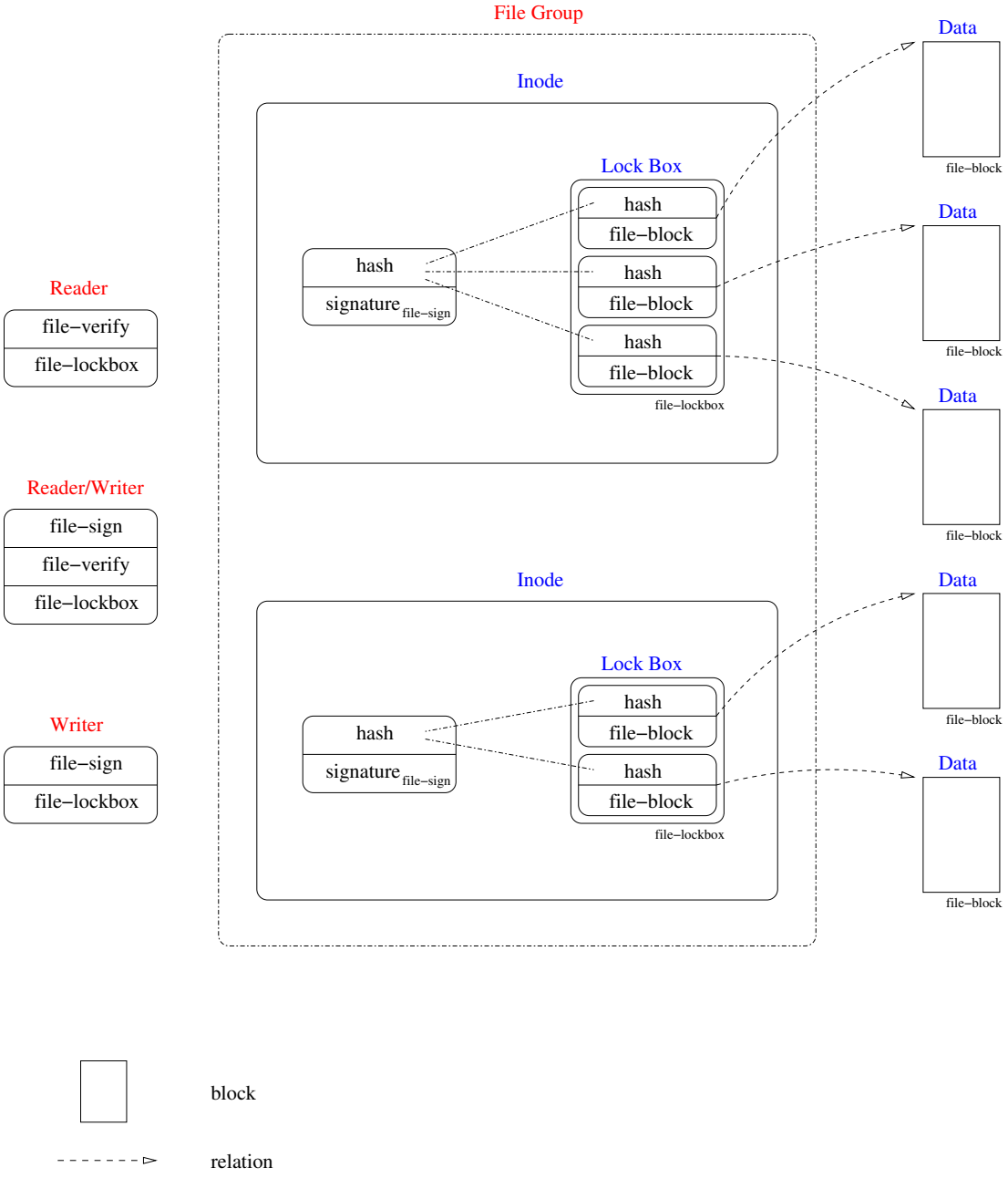


Figure 2.15: The *Plutus*' keys, locks and groups

Regarding access control management, *Plutus* makes use of lazy revocation. Indeed, re-encrypting the file's contents whenever a user is revoked would incur a large performance overhead. Instead, the re-encryption is delayed until the file is modified. *Plutus* relies on key rotation [FKK06] to address the issues of lazy revocation in the context of file groups.

Although *Plutus* aggregation of files according to their access control rules is ex-

tremely interesting, the overlay network has never been described. Besides, *Plutus*' access control scheme may lack flexibility when it comes to managing hundreds, thousands or millions of users since the system does not provide any mechanism for aggregating users into groups, for instance, which would greatly ease access control management, especially in large-scale peer-to-peer file systems.

2.3.7 Pastis

Pastis [mBPS05, Bus07] is a large-scale read/write peer-to-peer file system. *Pastis* relies on the *PAST* [RD01b] distributed hash table built upon the *Pastry* [RD01a] overlay network.

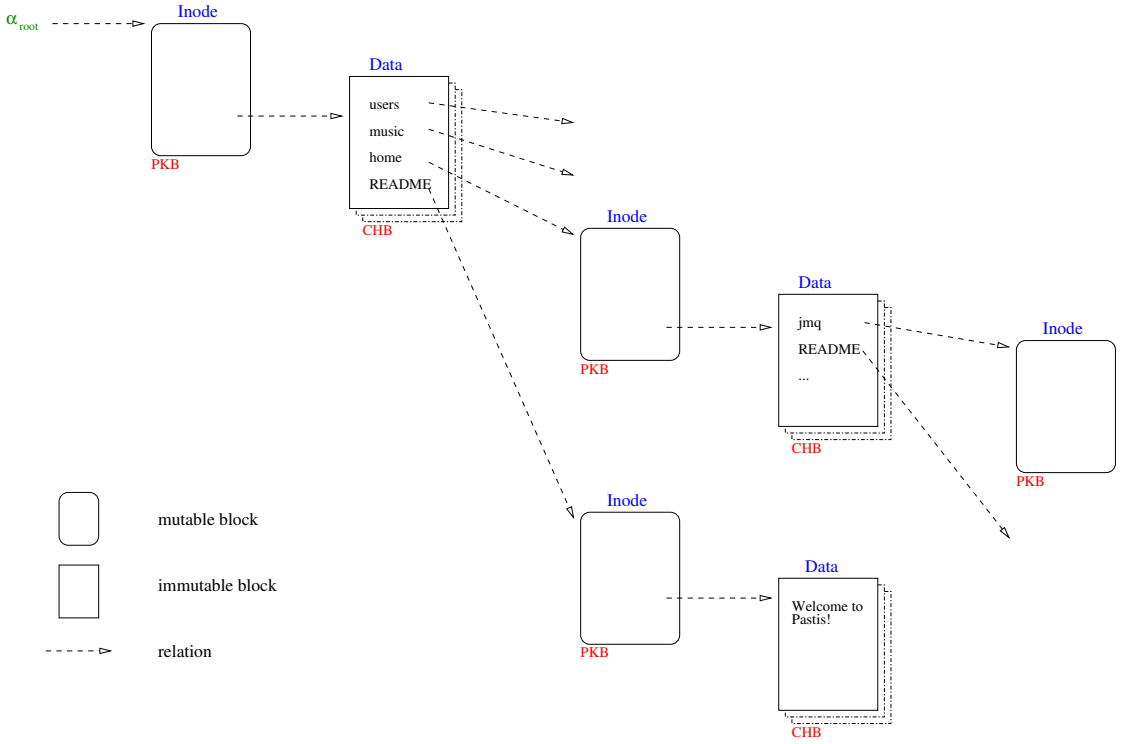


Figure 2.16: The *Pastis* organisation

Pastis follows *CFS*'s architecture but introduces mutable-block-based metadata in order to overcome *CFS*'s major issues. Indeed, *CFS* is said to be a read-only file system because it can only be modified by the administrator. Besides, since its architecture is based on immutable blocks, whenever an object is modified, the hierarchy must be updated up to the root block, which is then re-signed by the administrator. *Pastis* introduces *PKB*s along the way so that modifying an object only implies creating new immutable data blocks as well as updating the mutable metadata block. Since the number of blocks modified is independent of the hierarchy depth, *Pastis*'s design is far more efficient than *CFS*'s. In addition, since *PKB*s can

only be updated by a user possessing the private key required to re-sign the content, a *Pastis* file system can be modified by multiple users *i.e.* every object can be updated by its owner.

Figure 2.16 illustrates the *Pastis*' *UFS*-like hierarchical organisation composed of both metadata *PKB*s, known as *inode*s, and data *CHB*s. The figure depicts the use of *PKB*s which stops the update propagations.

Since creating a file system object in *Pastis* implies creating a *PKB*, the user may end up keeping a lot of cryptographic key pairs. In order to avoid storing all those sensitive keys, *PKB*s embed both the public key of the block owner and a signature of that key done with the block's generated private key. Therefore, the user does not need to keep any keys except her own key pair. Then, every operation performed on an object is signed with the user's private key instead of the block's private key.

Pastis also provides a write-only certificate-based access control mechanism. Any user willing to grant the permission to modify one of her objects can generate a certificate and distribute it to the authorised users. Then, whenever such a user performs a modification on the given object, she attaches the certificate to the *inode*'s *PKB*, proving that this operation is legitimate. Thus, a client retrieving the *inode* can verify that both the block and the certificate are valid. A certificate embeds the *inode* block's public key, the authorised user public key as well as an expiration date, the whole being signed by the object owner.

Pastis therefore provides both the access control and consistency models required to build a usable file system in a peer-to-peer environment. However, the access control scheme still suffers from the number of certificates the user must keep.

⋆ ⋆
⋆

This section intended to detail the internals of the major peer-to-peer file systems developed over the last decades and to give an overview of the trade-offs involved in the design of such systems. Research was also conducted in order to improve peer-to-peer file systems in alternative ways: *Chefs* [Fu05] is an access-controlled content distribution network built upon *SFSRO (SFS Read-Only)* [FKM02], *Total Recall* [BTC+04] is a system predicting hosts availability in order to optimise replica placement, *TFS (Transparent File System)* [CCB07] is a transparent layer which makes use of the unused local storage until the local operating system claims it and overwrites the cached peer-to-peer data and *Glacier* [HMD05] is a storage system relying on erasure codes in order to increase availability and durability.

Chapter 3

Environment

This chapter starts by defining the properties end-users expect a modern storage system to provide before carefully defining a model capable of guaranteeing them all. Finally, the objectives and assumptions of this work are defined according to the background discussed in *Chapter 2*.

3.1 Properties

This section discusses the properties end-users expect from a modern storage system. Although the properties below have been defined with the objective of designing an ideal storage system, some may seem more desirable than mandatory: anonymity, mobility, transparency, capacity or even cost for instance.

Durability

Durability ensures that once the system has agreed on storing some data blocks, those blocks will never be lost.

A system lacking this property would be incapable of guaranteeing the user to eventually retrieve her files. Commodity hardware such as hard disks but also external backups fail to ensure this property.

It is actually impossible to ensure durability by relying on a single instance of the data because the hardware storing this instance could be destroyed, stolen *etc.* Therefore, reliable systems tend to rely on redundancy such as replication to guarantee durability.

Noteworthy is that peer-to-peer file sharing applications such as *Bittorrent* [Coh03] actually lack this property. Indeed, such systems make use of pervasive replication since every client retrieving a file implicitly creates a new replica which can be used

to serve other client requests. Therefore, while popular content achieves a high replication ratio, rarely accessed files eventually get lost. This lack unfortunately makes the system difficult to use for users seeking unpopular content.

Integrity

A system providing integrity ensures that a client retrieving a block of data will end up with the exact content that has been previously inserted. Therefore, this property also guarantees that the data has not been altered in any way.

The property is usually provided through the use of integrity codes such as cryptographic hash functions, authentication codes or digital signatures.

Although most distributed systems perform the integrity verification process on the server handling the client's request, the client itself should be able to verify the block's integrity if given enough information such as the identity of the user emitting the authentication code for instance.

Availability

The availability property ensures that a data block stored by the system remains accessible at all times. This property coupled with the durability property makes the system reliable.

As for durability, replication is a way of achieving availability by maintaining a replication ratio such that even if some replication nodes fail, enough replicas remain in the system for the clients to access the data.

Note that applications relying on pervasive replication, such as *Bittorrent* [Coh03], can ensure neither the durability nor the availability property because the replication ratio depends on the content's popularity.

Privacy

The privacy property ensures the user the possibility to keep her files completely private, both from other users and more powerful entities such as the potential organisation distributing the software, the user's *ISP (Internet Service Provider)* or even governments *etc.*

Privacy is usually provided by means of cryptography: every stored data block is first encrypted on the client side so that the servers never have access to the data in its plain form.

Sharing

As described in *Section 1.1* sharing has become increasingly important to *Internet* users.

A viable system would therefore provide users means to share data with other specific users in an easy way.

In order to prevent both the organisation running the system and unauthorised users from accessing the data, systems usually distribute the cryptographic key used for encrypting the data to the users granted access.

Anonymity

People are usually concerned about companies or govenment entities analysing users' doings on the *Internet*. Indeed, people expect the same rights they are granted when it comes to their home privacy for instance.

Anonymity should therefore be guaranteed by the system such that nobody, not even the government or the user's *ISP* can know what data the user is storing, sharing or accessing.

Although anonymity has been studied through various projects [CSWH01, DFM01], the mechanisms used for providing such a guarantee are often very expensive and therefore impact the user experience.

Versioning

Computers have become the ultimate tool for treating information. As such, every document evolves in its digital form from one version to the next.

However, users may wish to undo a modification or rollback to a past version. Such a feature is considered fundamental in revision control systems such as *CVS (Concurrent Control System)*, *Subversion* and *Git*. Besides some storage systems [SFH+99] have started integrating such a functionality at the file system level.

Therefore, any modern storage system should provide a way for users to track the modifications applied onto documents but also to naviguate through the versions and potentially restore a specific one.

Mobility

With the increasing diversity in mobile devices, people are trading their old single desktop computer for a variety of small nomad devices from mobile phones to netbooks to tablets and so forth.

A viable platform for storing, sharing and synchronising files should be accessible from all these mobile devices by coping with the characteristics of such resource-limited computers.

Organisation

Since the advent of personal computers, people have been used to manipulate information through the abstraction known as the *file*. In addition, the hierarchical organisation consisting of a directory containing files and sub-directories has made its way to the general public as the traditional way for organising information.

However, recent research along with some commercial products tried to introduce another way for retrieving and searching documents through tags. Although this scheme has not yet supplanted the hiearchical organisation on personal computers, it appears to be a serious alternative.

No matter which scheme a storage system uses, the user requires a way for organising and managing her files.

Transparency

The files stored by the system should be accessible in a transparent manner such that the end-user does not have to differentiate accessing a locally stored file from accessing a file stored through a remote system.

More precisely, the system should enable existing applications to manipulate the files stored through the given storage system as they did when stored on a local hard disk for instance.

Efficiency

The user experience is crucial, especially when it comes to accessing files that were supposedly stored locally and therefore quickly retrievable.

The system should therefore focus on giving the end-user the impression that accessing files residing on other computers through the *Internet* is actually *"as fast as"* accessing them locally.

The user could well be aware of the fact that the network protocols impact the performance of the system especially regarding the network latency since nodes may be geographically far from each other. However, the networking aspect of the system should not make the user's common operations a hassle, such as watching a movie, working on an office document, listening to music *etc.*

Various techniques could be used for achieving efficiency from optimised network protocols [Coh03] to caching algorithms for instance.

Capacity

The user should not be limited regarding the number of files or the size of the files that she can store and should have access to a storage capacity of the same order of magnitude as the capacity offered by her local hard disk.

Indeed, given the evolution of the hard disk prices, user may lack incentive to move on to a reliable, secure and available storage infrastructure if one can get twice as much storage capacity by buying cheap external drives from a nearby hardware store.

Cost

Along with the capacity property, the cost of such a system should be low for the client wishing to use it.

Besides, the costs for the organisation developing and maintaining the infrastructure should also be as low as possible because such costs would have to be passed on to the consumer, one way or the other.

3.2 Model

This thesis claims that a file system abstraction on top of a peer-to-peer network is the most suitable model for achieving the fundamental properties defined above.

3.2.1 File System

End-users have been accustomed to hierarchical organisations since the introduction of the file system paradigm. Providing the user a similar way to organise files is crucial. Although most storage services and products [Box, Dro, Win] provide such a hierarchical organisation, some still put the user in front of a flat name space. Peer-to-peer file sharing applications [Coh03, CSWH01, DFM01] for instance fail to offer users a hierarchical organisation making it difficult for people to organise the files they contribute to the system but also to browse other users' contributions.

Although the organisation property is fundamental, transparency is also extremely important. Much storage software [Box, Dro] forces the user to use a specific application. It may, at first, seem natural from the system designer's perspective because

defining a specific interface gives the application the liberty to interact directly with the end-user but also to offer features specially designed for this system. However, such systems also suffer from it since breaking the compatibility with all the applications relying on the file system interface automatically isolates the software from the rest of the world. Indeed, applications would not be able to use files stored through a system incompatible with the standard file system interface meaning that users would not be able to play their music files or watch their movies, they would have to first retrieve the file from the network, store it on the local disk before the application could proceed and open it.

Thus, in order to respect the organisation and transparency properties, and according to end-users habits, such a storage infrastructure should be accessible through a standard file system interface.

3.2.2 Peer-to-Peer

As studies [DB99, BDET00, BDET00, HAY+05] suggest, file systems can benefit from the peer-to-peer architecture in a number of ways.

Peer-to-peer systems offer a way to aggregate and make use of the resources on computers across the network hence building a virtually infinite and highly adaptable system. Research showed [Vog99, DB99] that the usage of computers' storage oscillates between 53% and 87% meaning that a large portion of the local storage space is, most of the time, unused. This storage characteristic indicates that peer-to-peer networks can ensure the durability, availability, versioning and capacity properties by making use of a user's unused space for replicating the other users' data. Furthermore, by relying on the clients for contributing the system in bandwidth and storage capacity, the costs for running such a system are kept extremely low, both for the organisation running the software and the end-user.

As shown in *Chapter 2*, peer-to-peer overlay networks, more specifically structured overlay networks, have been designed to be highly scalable. However, although this characteristic implies that the load put on the nodes depends on the size of the network, it does not guarantee that mobile devices, for instance, will have enough resources to support such a load. Indeed, a user might want to contribute the peer-to-peer network from her home desktop computer only, while accessing it from multiple other devices, not mentioning that such resources-limited devices may not have the capacity to maintain the local network state. Fortunately, structured overlay networks have been designed to be highly tunable through several parameters. For instance, the *Chord* [SMK+01] overlay network's base parameter can be chosen in order to achieve the desired trade-off between lookup performance and the size of the local state every node must maintain.

The large-scale nature of peer-to-peer networks also contributes to improving the overall system performance. Indeed, since data can be distributed throughout the network but also retrieved from multiple nodes at the same time, the bandwidth load is naturally balanced between the computers contributing to the system. The *Bittorrent* [Coh03] peer-to-peer file sharing application gained in popularity due to its efficient network protocol which makes use of this characteristic.

The peer-to-peer model therefore appears as a natural network paradigm for ensuring most of the system's properties such as durability, availability, versioning, mobility, efficiency, capacity and cost.

3.3 Mission

Peer-to-peer networks have been shown to exhibit many interesting characteristics but also introduce many challenges. This thesis does not discuss the challenges related to overlay networks or even distributed hash tables because a substantial amount of work has already been achieved in these fields, as attested by *Chapter 2*. Topics ignored by this work therefore include, but are not limited to, redundancy algorithms [HAF10], consistency models, agreement protocols, fault tolerance, atomicity, garbage collection [BCK+09], overlay network's identifiers assignment [FJG06], mutual exclusion algorithms [MCG05] and routing algorithms [HCW10, dALF10, HB11].

Unlike centralised facilities which are very expensive to build and maintain, peer-to-peer systems do not require any special administrative or financial arrangements. Such systems therefore became very popular for exchanging information freely, outside any control. Research in anonymity arose as an additional step to freedom on the *Internet*, led by well-known projects such as *FreeNet* [CSWH01] and *FreeHaven* [DFM01]. Although anonymity may be considered by many as a fundamental requirement in today's digital world, this research topic will not be discussed in this thesis and is left as future work [CLL07, ZSJ06, Mha11].

Likewise, the versioning [CRS05, JXY07] feature is not studied in this thesis. As such, the underlying distributed hash table is assumed to store the latest version of every block.

Since overlay networks and distributed hash tables have been the focus of the research community for more than a decade, this work concentrates on providing file system functionalities in a decentralised, hence untrustworthy environment.

The file system component, built on top of a block storage layer, provides the following fundamental functionalities. First, file systems introduce the *file* abstraction: a block of arbitrary information. Then, files are associated with a human-readable

identifier, known as *path* in hierarchical systems, forming a tree-like organisation scheme. Third, the notion of *user* distinguishes the multiple entities interacting with the file system. Finally, the access control scheme enables users to control their files, directories *etc.* enabling files to be shared with or protected from other entities.

File systems integrate another inherent but non-obvious functionality. Although most file systems' operations relate to objects such as directories, files and links, a few operate on the whole system configuration. Centralised file systems tend to rely on a specific user, known as *root* on *UNIX*-like systems and *Administrator* on *Windows*, to perform such special operations. This user, being granted super-privileges, can perform system-wide actions such as creating users and groups, accessing or removing any file but also modifying the file system metadata such as its name, its capacity along with some specific parameters. Decentralised file systems however, cannot rely on such a special user because such systems were specifically designed to prevent a single entity from controlling the whole system. Therefore, the administrator entity must be re-considered to fit such a decentralised environment.

This thesis focuses on designing a flexible access control scheme for decentralised untrustworthy storage environments, providing peer-to-peer file systems' users a way to control their files individually. In addition, an administration scheme is discussed which both prevents a single user from completely controlling the system and enables users to request an administrative operation such that, if it is beneficial to the system, it will be carried out.

Although the community showed great interest in such distributed systems, rare [HAY+05] are the decentralised file systems to have been deployed. The final objective of this thesis is to develop a viable prototype proving feasible the deployment of such a system to a large number of users in a production environment.

3.4 Assumptions

The file system described in this document relies upon a distributed hash table, which in turn, is built on a peer-to-peer overlay network. Although the challenges related to overlay networks and distributed hash tables are not discussed throughout this thesis, several assumptions are made regarding the interface of the underlying storage layer but also the properties and guarantees of the network architecture.

First, the peer-to-peer network is assumed to be untrustworthy. Indeed, since such networks are mostly populated by personal computers, no assumption can be made regarding the trustfulness of the contributing nodes. Furthermore, the decentralised nature of such networks coupled with the untrustworthy assumption implies that nodes must operate in a completely symmetric way. Indeed, since peer-to-peer nodes

are considered equally unprivileged, everything performed by one node could also be performed by another one. In addition to those fundamental characteristics—decentralisation, untrustworthiness and symmetry— the high dynamicity of such networks requires protocols to be scalable. Finally, the inherent network's churn implies that no assumption should be made regarding nodes' connectivity.

Second, in order to ensure the durability and availability properties, redundancy has been shown to be an absolute requirement. Projects such as *OceanStore* [KBC+00] and *FARSITE* [ABC+02] have been using agreement algorithms such as the *BFT* [CL99] protocol and *Paxos* [Lam98] in order to ensure consistency among the replicas. Unfortunately, such algorithms are known to be expensive [DW01, Bus07] as detailed in *Section 2.2*. Other projects such as *CFS* [DKK+01], *Ivy* [MMGC02] and *Pastis* [mBPS05] chose to rely on quorums. Indeed, since peer-to-peer file systems only require to store and retrieve data, such algorithms achieve better performance than their agreement counterparts. According to the efficiency property, the underlying distributed hash table will therefore be assumed to be making use of quorums for maintaining the replicas in a consistent state.

As detailed in *Section 2.2*, quorum-based Byzantine systems replicate every object on $\varphi \geq 3\gamma + 1$ storage nodes in order to tolerate up to γ malicious nodes while read and write quorums must contain contain, at least, $2\gamma + 1$ replicas. Note however that every client must be able to distinguish the illegitimate replicas provided by Byzantine nodes. Therefore, and since the untrustworthy property implies that storage nodes cannot be trusted, every data item must be self-certified that is, every data block must include the necessary information in order to ensure that *(i)* the block corresponds to its supposed address *(ii)* the block's integrity has been maintained and *(iii)* the block's authenticity is guaranteed. The self-certification property along with the replication and symmetry ensures that any client can select the valid instance from a set of replicas.

Third, the distributed hash table should provide an interface composed of, at least, the four fundamental routines below.

- `Put`(α, β)
- `Get`$(\alpha) \longrightarrow \beta$
- `Gather`$(\alpha) \longrightarrow \beta$
- `Erase`(α)

The following details some of the essential distributed hash table's protocols. Since these protocols reflect the particularities—untrustworthiness, symmetry and self-certification—of the given environment, understanding those protocols will help the reader comprehend the design decisions made in *Chapter 4*.

Put

The `Put`(α, β) routine takes a unique address α along with the data block β to be stored. In order to ensure durability and availability, the block is replicated on a set of nodes Ω such that $|\Omega| = \varphi$. The process of storing a block in the distributed hash table goes as follows.

The client starts by sealing the block in order to ensure self-certification. Indeed, since the block is going to be stored on untrustworthy nodes, future clients retrieving the block must be able to detect blocks that have been illegally altered by malicious nodes. Integrity can be ensured through the use use of a cryptographic signature, a *MAC (Message Authentication Code)* [OM94] or equivalent.

The client then computes the block's address and invokes the `Put`() routine which locates the nodes Ω responsible for the given address α. The block β is sent to a write quorum of storage nodes after which the client waits for their responses. Once $2\gamma + 1$ acknowledgements have been received, the block is considered as being stored by the system.

From the storage node perspective, the process consists in verifying the received block's validity before storing it. The self-certification process is composed of several steps. Firstly, the node verifies that the address α corresponds to the block β. Note that this implies that the block's address computation must be a function of the block itself. Secondly, the block's integrity and authenticity is checked, by verifying the embedded cryptographic signature for instance.

Note that the Ω storage nodes periodically synchronise with each other in order to maintain replica consistency. Besides, write operations performed on quorums are actually composed of two phases. The first phase consists in locking the quorum nodes in order to ensure mutual exclusion. In the second phase, the client sends the block's content to the quorum nodes. However, for the sake of simplicity and clarity, the first phase is ignored and will therefore not be discussed throughout this thesis.

Get

The `Get`(α) routine is used to retrieve the block identified by α. More specifically, this routine returns to the client the first valid block instance located throughout the network. Therefore, depending on the implementation and the context, the block may be retrieved from one of the Ω nodes, from a node contributing to the routine process and having the block in cache, or even from the client's cache.

Once the client has received the block, it proceeds to the exact same verification process as the servers: *(i)* the received block β corresponds to the requested address α and *(ii)* the block's integrity and authenticity is maintained. The reader should

notice the symmetry of the nodes' behaviour. Indeed, the verifications performed by a node acting as a server will eventually be performed by a client. Therefore, the information necessary to enact a block's validity must be available to every node, being a client or a server, hence the self-certification.

Gather

The `Get`(α) routine's particularity lies in the fact that the first valid instance found is returned. Therefore, this routine is particularly interesting for retrieving immutable blocks. Indeed, since such blocks do not evolve over time, the client cannot end up with an incorrect version as a single version of this block will exist for ever. Therefore, retrieving an immutable block which passes the validity tests is sufficient to affirm that this block is the one the client is seeking.

However, such a routine would not be satisfactory should an application make use of mutable blocks. By using the `Get`(α) method, a client could end up with a block which is valid but happens to represent an older version of the requested block. Indeed, since data dissemination in computer networks is by definition asynchronous, hence unreliable and non-atomic, multiple nodes may, at the same time, store or cache different versions of the same block.

The `Gather`(α) routine addresses this issue by directly requesting the block from Ω through the formation of a read quorum composed of $2\gamma + 1$ storage nodes. Once $2\gamma + 1$ instances of the block have been received, the client starts by discarding any invalid block *i.e.* violating the integrity and/or authenticity for instance. Since the algorithm has been designed to tolerate up to γ Byzantine nodes, the number of such invalid blocks should not exceed γ. Finally, among the remaining instances, the client picks the one with the highest version number. Indeed, mutable blocks are expected to embed a version number in order to differentiate the multiple variations of a given block α. Besides, note that a storage node being requested to overwrite a mutable block would verify that the version number of the new block is strictly higher than the currently stored one. This additional verification step ensures mutable blocks evolve in a monotonic way.

Figure 3.1 illustrates the protocol in a network tolerating up to $\gamma = 2$ Byzantine nodes. The system initial state is consistent since the Ω nodes store the exact same version of the block *i.e.* version 3, except for the Byzantine nodes whose behaviour cannot be predicted. Then, a client modifies the block by updating a write quorum containing both malicious nodes. Finally, another client requests a read quorum which contains the Byzantine nodes and the two nodes that were not included in the previous write quorum. However, since the algorithm ensures read and write quorums intersection, the client can discard the invalid blocks and pick the latest

version among the three remaining instances which happens to be version 4, as expected.

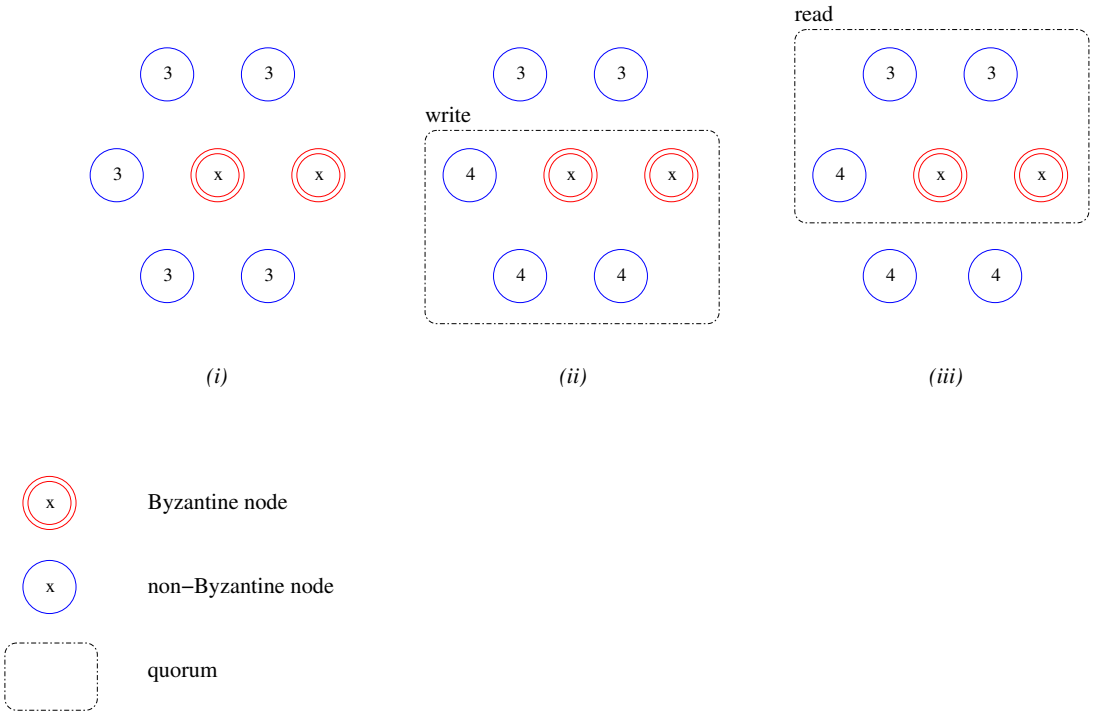


Figure 3.1: A three-step representation of a symmetric quorum-based system

The following example, illustrated by *Figure 3.2*, shows that symmetry is absolutely crucial to decentralised untrustworthy environments. Let us consider a distributed hash table which, for some optimisation purposes, authorises clients to store data between 2 *a.m.* and 4 *a.m.* only. A client would start by sending its block to a write quorum of Ω. The storage nodes would verify that *(i)* the block is valid and *(ii)* the request has been made between 2 *a.m.* and 4 *a.m.*, in other words, the current time lies in this interval. Then, let us consider the Byzantine node $\omega \in \Omega$ which stores one of the block's replicas. This node decides, although it is 5 *p.m.*, to modify the block. However, instead of following the protocol, it just replaces the local replica with its new version and does not bother contacting the other Ω nodes. Considering that the block is valid and has the highest version number, a client wishing to retrieve the block could form a read quorum including ω. Therefore, the client would end up with, say, up to $\gamma - 1$ invalid instances of the block, $\gamma + 1$ valid instances of the block and one valid, though illegally forged, instance of the block embedding the highest version number. The client, following the protocol, would therefore discard the invalid instances and keep the latest version, version 5 in the example, which happens to be an illegal instance.

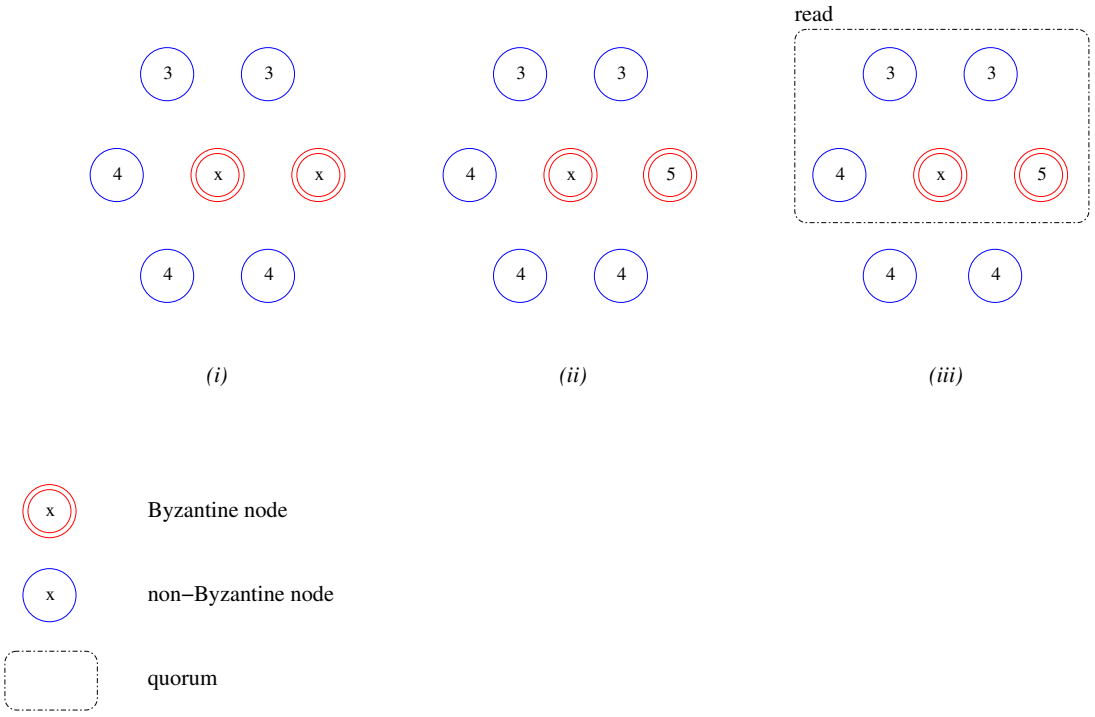


Figure 3.2: A three-step representation of an asymmetric quorum-based system

This example illustrates the system's lack of symmetry. Indeed, the validation process is not symmetric because it cannot be performed by clients since they do not have access to the time the block has been stored. Therefore, the clients have no choice but to trust the servers regarding this predicament. Since the environment is assumed to be untrustworthy, the assumption is violated and the system is flawed.

Noteworthy is that the `Get`(α) and `Gather`(α) methods are distinguished in this document for the sake of clarity. However, most implementations, including the one discussed in *Chapter 5*, merge both functionalities into a single routine which, depending on the address α, operates in one or the other mode.

Erase

The `Erase`(α) routine takes the address of a block that must be removed from the distributed hash table. The client wishing to perform such a removal starts by sending a request to the Ω nodes storing a replica of the block. Storage nodes receiving such a request then challenge the user in order to verify the legitimacy of the operation. Once the client has been authentified as the creator, every replica is destroyed. Note however that the authentication process depends on the type of block, as described in *Chapter 4*.

⋆ ⋆
⋆

To summarise, this thesis makes several assumptions among which are the distributed hash table's properties: decentralisation, untrustworthiness, symmetry, self-certification, scalability, replication through quorum protocols and non-connectivity.

Considering such a distributed hash table, this thesis aims at designing a decentralised peer-to-peer file system by focusing on providing a flexible access control scheme along with a mechanism for users to request administration tasks in a system devoid of any authoritative entity.

Chapter 4

Design

This chapter details the design of the fundamental components of a peer-to-peer file system.

The first section focuses on describing the access control scheme which introduces both the notion of user and the abstract representation of file system objects such as files, directories *etc.* This section starts by defining the objectives of the access control scheme given the environment described in *Chapter 3*. The model of the access control scheme is then discussed though this model will be refined later one. Finally, the concept behind the access control mechanism is introduced before discussing in details the internal representation of the blocks composing the file system hierarchy.

The second section discusses the file system organisation and the necessity for enabling users to perform administrative operations. First, the semantics of centralised file systems are discussed in the context of peer-to-peer file systems. A model is then proposed for both administering the file system and transferring object ownership. The design of the proposed model is then detailed through the introduction of a new physical block.

By the end of this chapter, the fundamental components will have been designed, leading the way to the implementation of a viable peer-to-peer file system prototype.

4.1 Access Control

Although many decentralised peer-to-peer file systems have emerged in the last decade, none of them succeeded in providing users with a flexible access control system.

The well-known file sharing applications such as *Bittorrent* [Coh03], *Freenet* [CSWH01], *FreeHaven* [DFM01] *etc.* actually provide a content distribution infrastructure more

than a way of sharing files since users cannot decide which users are allowed to retrieve the documents they contribute to the system. Although users may not be interested in controlling access to movies, they might be for more personal information such as family photos, work documents and so forth.

While some distributed file system projects [DKK+01, MMGC02] lack an access control mechanism, others such as *OceanStore* [KBC+00] and *FARSITE* [ABC+02] do provide privacy control functionalities. Unfortunately, such systems suffer from fundamental flaws regarding our target environment such as the non-scalability of the network architecture or the use of expensive algorithms, as explained in *Chapter 2*.

More recently, several research projects, including *Chefs* [Fu05], *Plutus* [KRS+03] and *Pastis* [mBPS05], focused on access control in untrustworthy environments.

Chefs' single-writer/multi-reader design might well suit content distribution applications such as *Bittorrent* [Coh03] but unfortunately lacks flexibility when it comes to large-scale file systems. On the contrary, *Plutus* provides a multi-writer/multi-reader scheme but, like *Chefs*, requires the users to be connected whenever an object's owner wishes to grant them access. This connectivity requirement is unpractical for large-scale networks where the churn rate has been measured to be very high [LSG+04]. *Plutus* also puts some trust constraints on the storage nodes handling write operations, hence, violating the untrustworthiness predicate. Finally, *Pastis*' access control scheme, very much like *Plutus*', constrains the users in keeping a constantly growing number of certificates and cryptographic keys.

Noteworthy is that the most recent work achieved through *Chefs*, *Plutus* and *Pastis* indicate that issues remain to be addressed. The remainder of this section therefore presents the design of a flexible access control scheme for the given environment which does not require users to keep any access information but their identity *i.e.* a single cryptographic key pair.

4.1.1 Objectives

The following statements define the scope within which the access control scheme has been designed along with the characteristics such a mechanism should incorporate. Although one might disagree with these definitions, this set of rules has been defined in order to provide the access control mechanism functionalities common to most file systems while taking the environment's particularities into consideration.

∇_1 First, the environment characteristics and fundamental properties defined in *Chapter 3* must be respected throughout the design process. These include decentralisation, scalability, untrustworthiness, symmetry and self-certification

but also non-connectivity due to churn, efficiency through quorum protocols and so on;

∇_2 A user must be able to modify its object's permissions. Furthermore, the effects of those modifications should be made effective immediately. Although atomicity is obviously unachievable given the asynchrony of the underlying physical network, this rule suggests that any operation that does not comply with the object's last set of access control rules should be rejected;

∇_3 Any user should be able to consult an object's current permissions. Note that this rule conflicts with most common file systems which prevent users from collecting information on inaccessible objects. However, in the given context, the lack of a centralised entity makes it difficult to prevent a user from retrieving the block corresponding to the file's metadata—assuming she knows the block's address—hence accessing its access control information;

∇_4 *Section 4.1* showed that the most recent research regarding access control in decentralised environments suffered from the amount of access information users have to keep locally. Indeed, both *Plutus* [KRS+03] and *Pastis* [mBPS05] require clients to store a linearly increasing number of keys and certificates. The storage space required on clients for managing objects' access control should therefore be ideally reduced to a single item;

∇_5 The large-scale environment's characteristic implies an extremely large and dynamic number of users and files. While common centralised file systems such as *ext2 (Second Extended File System)* were designed with space consumption and simplicity in mind, decentralised file systems must provide flexible capabilities for users to manage the possibly thousands of users having been granted access to an object. The access control scheme should therefore enable users to create hierarchical *group*s which, as the name suggests, can be composed of both users and/or sub-groups. This paradigm would enable users to organise their friends, acquaintances, family *etc.* hence easing the access control management;

∇_6 According to the environment specifics and with regard to ∇_5, the access control scheme should be as efficient as possible. Especially, the complexity of the process consisting in verifying that a user's operation is legitimate should be logarithmic, if not constant time; and

∇_7 Since data retrieval cannot be controlled, anyone is allowed to request a data block from the underlying distributed hash table. Therefore, accountability regarding users accessing data seems unachievable. However, users should not be capable of repudiation regarding object modification.

Note that every access control scheme candidate will be considered unsuitable if violating at least one of these objectives. Besides, whenever such an objective is mentioned through its ∇_n symbolic name, the reader will be able to refer himself to the definitions summary located at the bottom of the page.

4.1.2 Model

The following discusses the particularites of access control schemes in the given environment.

4.1.2.1 Policy

An access control system is one which enables an authority to control the access to resources. In the context of file systems, access control systems enable a user to grant a set of other users access permissions onto an object, being a file, directory, link *etc.*

Access control systems are often categorised as either discretionary or non-discretionary, the most widely recognised models being *MAC (Mandatory Access Control)* [Cla83], *DAC (Discretionary Access Control)* [Kar86] and *RBAC (Role-Based Access Control)* [JB94, Vas08].

MAC is an access control policy determined by the system, through an authoritative entity. Historically, *MAC* has been designed and used by military organisations processing highly sensitive data. In such systems, subjects and objects are assigned a label so that a user can access a document only if her clearance level is equal or higher than the document's sensitivity level.

DAC is an access policy determined by the object's owner. Therefore, the user decides who has access to the object and what operations they are allowed to perform. Note that unlike *MAC*, *DAC* models do not require any authoritative entity.

Finally, *RBAC* is an alternative approach consisting in the definition of various roles matching the multiple organisation's personnel functions. The permissions to perform certain operations are then assigned to roles. Finally, every member of the personnel is assigned a particular set of roles such that, through those assignments, the user acquires the permissions associated with the roles. The *RBAC* model simplifies the whole access control management since controlling the access policies consists in assigning roles to individuals.

As discussed in *Chapter 3*, a consequence of the peer-to-peer environment is that no entity has complete control over the whole system. For this reason, both *MAC* and *RBAC* models, which require system-wide definitions, cannot be used in this context. Therefore, in order not to violate ∇_1, the *DAC* model must be used.

4.1.2.2 Pattern

Systems such as operating systems, file systems, websites *etc.* are said to make use of *active* access control because permission is granted at the time an operation is requested or performed. Most systems follow this pattern because all requests are made to a manager which decides whether or not to grant access. Since access control information is centralised, the manager can easily take such a decision.

On the other hand, distributed systems tend to dynamically build managers by relying on Byzantine agreement protocols. Unfortunately, and as discussed in *Chapter 3*, since they reduce concurrency [DW01, Bus07], such algorithms are impractical for many applications, especially large-scale distributed file systems.

As a consequence, the given environment cannot make use of managers. Access control is therefore said to be *passive*. The idea behind passive access control is to store access control information along with the object so that any client retrieving an object can verify that the last modification has been performed by a legitimate writer *i.e.* the writer had the permission at the time, τ, the operation was carried out. If, as described through *Section 3.4*, the block happens to be illegitimate, it is discarded until a valid instance is found.

Thus, users writing an object must attach an *atemporal* proof such that, at any later time, anyone can verify that the object has been properly constructed according to the permissions in place at τ. By doing so, the system's symmetry is maintained and ∇_1 is respected. Furthermore, and in order to prevent violating ∇_7, the proof should enable users to identify the writer.

Regarding read operations, since the storage nodes cannot be trusted, the objects' content should always be encrypted. The access control scheme should therefore enable objects' owners to distribute the key to authorised readers while respecting ∇_2 and ∇_4.

4.1.2.3 Class

Access control schemes basically fall into one or both of the two following classes: token-based and record-based. This section takes both of the access control classes and shows that no scheme can achieve the required properties in the given environment.

An implication of ∇_5 is that permissions must be flexibly manageable through hierachical groups, giving the user a tool for organising users very much as a tree-like file system view enables users to organise their files.

As mentioned in the previous section, a passive access control model implies that users must attach to the object a proof showing evidence of the legitimacy of the

∇_1: environment, ∇_2: modification, ∇_3: consultation, ∇_4: space, ∇_5: flexibility, ∇_6: efficiency, ∇_7: accountability

operation. In more practical terms and with consideration of group hierarchies, every proof involving one or more of those groups will be composed of sub-proofs, each providing evidence of group membership at τ, the whole forming a *chain* of proofs.

Token-Based

In token-based access control schemes, objects' owners distribute unforgeable tokens to clients, granting them the permission to perform operations, while nothing is kept on the manager's side except what is strictly necessary to verify tokens' validity. *Certification* and *Capabilities*, for instance, fall into this category.

In active access control models, clients pass their token to the manager. If the chain of tokens is valid, the requested operation is accepted. In a passive scheme, the user attaches a chain of tokens to the object to be modified so that nodes retrieving the object can verify that the writer provided a proof of her legitimate action.

Note that since everybody must be able to verify the tokens' validity, such tokens must be protected from public disclosure, for example by securely identifying its holder. *Certification* schemes, for instance, include the user's identity in a digital signature for ensuring this property. Additionally, such a user identification complies with ∇_7.

A problem arises when it comes to verifying a proof. Indeed, to verify that a user had the permission—the tokens had neither expired nor been invalidated—at τ, the object must carry time-related information such as the time the object was updated. Unfortunately, even assuming that the system benefits from a globally synchronised clock, neither the storage servers nor the users can be trusted to provide a correct time. Indeed, malicious clients and servers could go back in time and claim a date that makes past tokens still valid. The solution would be to either rely on a centralised and trusted time server for digitally timestamping every update or to make use of consensus algorithms, both violating ∇_1 and ∇_6.

Record-Based

In record-based access control models, a subject's access depends on whether her identity is located in the records associated with the object. In active models, the manager keeps the records and performs the verification for every received request while, in passive models, an attached proof must provide evidence that, at τ, the user's identity could be located in the records of the groups she claimed to have been a member of. *ACL (Access Control List)*s, for instance, fall into this class of access control.

Unlike token-based access control models, access information is recorded in blocks,

∇_1: environment, ∇_2: modification, ∇_3: consultation, ∇_4: space, ∇_5: flexibility, ∇_6: efficiency, ∇_7: accountability

along with the other object's metadata. ∇_5 and ∇_3 imply that independent entities such as groups also record their access information in blocks.

Besides, objects and groups, in order to be accessible through the same address α at all times, must rely on mutable blocks because the address of such blocks remains the same while their content evolves.

Proving that a subject had the permission to perform the operation at τ comes down to proving there existed a link between the object and the subject, perhaps indirectly through several groups' memberships. Therefore, a client updating an object must attach a snapshot of the chain of groups, hence proving the existence, at τ, of a path from the object to the subject.

Unfortunately, since groups evolve over time, a group's block exists in different versions. Therefore, nothing could prevent a malicious user from using a past group's snapshot, at a time when she was a valid member.

Thus, as nodes could not be trusted to provide a valid timestamp in token-based models, servers and clients, once again, cannot be trusted to include, in the chain of proofs, the proper latest version of the groups' snapshot at τ. A malicious user could therefore go back in time by providing past versions of groups' snapshots, granting herself the permission to perform the operation.

4.1.3 Constraints

The previous section showed that any passive access control scheme violates the fundamental symmetry property.

However, one should notice that, by loosening constraints, it becomes possible to design such a scheme. For instance, *Plutus* [KRS+03] makes use of a token-based access control scheme where an object's owner distributes a key to the writers and the complement key to the readers. Since accountability is not a requirement, users are free to re-distribute the keys to whoever they wish. Therefore, *Plutus* requires users' connectivity for passing keys. Besides, since keys are freely distributed, nobody can consult the currently granted permissions.

The requirements of this work are therefore extremely strong compared to *Plutus*' and as a result, it has been shown impossible to achieve them all.

There is therefore no choice but to loosen the constraints in order to provide access control to peer-to-peer applications. Although ∇_3 and ∇_7 might seem questionable in terms of usefulness, especially in large-scale networks, connectivity is the environment property that the author believes is usually misinterpreted for the reasons exposed below.

- Research regarding churn rates in peer-to-peer networks has been performed on peer-to-peer file sharing applications such as *Bittorrent* [Coh03] because these are the only deployed large-scale applications that can be used to gather such information.

  However, the application itself has an impact on the node churn since, for example, users tend to stop sharing a file that has been downloaded to avoid wasting their upload bandwidth.

  Therefore, the author claims that different well-integrated systems such as file systems, instead of file sharing applications, would decrease the churn rate especially if users have incentives in contributing to the system's connectivity;

- Connectivity, very much like bandwidth, should increase as it has been the case since the advent of computer networks. On the other hand, new devices such as mobile phones, netbooks along with new user behaviours must be taken into consideration.

  One should note that although the number of such mobile devices is increasing extremely rapidly, most users possess multiple computers including a desktop computer at home and/or at work, a laptop, a mobile phone *etc.* Therefore, although the increasing mobility of computing devices implies nodes frequently joining and leaving the network, the user behind these devices is likely to be connected at all times through one or more of those devices; and

- Finally, although the probability of a specific user being connected might not be as high as expected because of the high churn rate measured in peer-to-peer networks, the probability of having at least one member in a set of users connected to the network should be higher, depending on the set cardinality.

Therefore, loosening the non-connectivity requirement will enable users, as in *Plutus*, to retrieve information from other users.

4.1.4 Concept

The following provides insights into the passive *ACL (Access Control List)*-based—*i.e. DAC* and record-based—access control scheme described throughout this section. Note that the record class has been chosen in order to prevent users from having to store access information such as certificates or keys as both *Plutus* [KRS+03] and *Pastis* [mBPS05] suffer from this characteristic.

The idea behind the presented access control scheme is to distinguish users according to their access relation to the target object. First, the users who have been granted access to the object directly by the owner are referred to as the *lord*s. These users

∇_1: environment, ∇_2: modification, ∇_3: consultation, ∇_4: space, ∇_5: flexibility, ∇_6: efficiency, ∇_7: accountability

play a particularly important role in the access control scheme because their access permissions can only be modified by the object owner. On the other hand, users who have been granted access to the object indirectly through one or more group memberships are referred to as the *vassal*s. The object owner has no direct control over these users since the access management has been delegated to the respective group managers. Therefore, vassals could join or leave groups that have been granted permissions on the object without the object owner even knowing.

The access control scheme's fundamental concept is to let lords access the object without additional constraints, assuming that they have been granted the appropriate permissions. However, the vassals are never given the key for decrypting the data, neither can they update the object directly. Rather, the idea is to rely on lords to vouch for the vassals by verifying that the requesting vassal has the permission to perform the operation. Assuming that the vassal does have the proper rights, the lord generates a certificate stating that, at τ, the vassal had been indirectly granted the permission to update the object. Likewise, a vassal wishing to read an object would need to contact a lord which would verify the vassal's permission before passing her the key for decrypting the data. Quite obviously, the object's owner as well as the group managers could also act as lords, hence vouch for vassals.

Noteworthy is that, although the users' connectivity is assumed to be higher than previously stated, only users accessing objects indirectly will need to contact other more privileged users. The lords' connectivity is therefore absolutely crucial to the system. Thus, object owners should make sure to grant access to several lords in order to ensure that the number of connected lords is sufficient to enable legitimate vassals to operate on the object. Should the number of such lords be insufficient, the application could warn the user for instance.

Finally, the access control scheme introduced hereby requires users to exchange information with one another. Therefore, as the overlay network enables nodes to route a message to the home node responsible for a given identifier, users now require the overlay network to provide a routine for locating a particular user. The rest of this document thus assumes that the underlying overlay network provides a `Locate`() method, which, given a set Ξ of user identities, returns the identity of a currently connected user. An easy way to provide such a functionality would be for the application to automatically set the user's, potentially multiple, *IP (Internet Protocol)* address in its associated *User* logical block, described next, such that given an identifier, one can easily contact the user by sending a message to her node.

4.1.5 Scheme

The access control scheme detailed in this section enables users to protect their objects against unauthorised read and write operations by granting permission directly to specific users and/or indirectly by delegating access control to third parties *i.e.* groups.

The following introduces the data block representations necessary to the access control mechanism, from physical blocks to logical blocks such as file system objects, users, groups and so forth. Every block representation is illustrated by a figure along with the three procedures below:

- The `Setup`$(\beta) \longrightarrow \alpha$ method is invoked whenever a block is built and returns the address of the freshly initialised block;
- The `Seal`(α, β) routine is called whenever the block has been modified and requires to be sealed, before being stored in the underlying distributed hash table for instance; and
- Finally, the `Validate`(α, β) procedure verifies that the given block is valid. This method is never explicitly invoked but rather used internally by the `Get`$(\alpha) \longrightarrow \beta$ and `Gather`$(\alpha) \longrightarrow \beta$ routines.

4.1.5.1 Physical Blocks

As in *CFS* [DKK+01], *Pastis* [mBPS05], *SFS* [MKKW99], *OceanStore* [KBC+00] and many other projects, the physical blocks are distinguished according to their immutability.

Content Hash Block

*CHB (Content Hash Block)*s are immutable blocks whose address is computed by applying a one-way function on the data. Thus, assuming the block is modified, a new content is implicitly created, hence generating a new address. *Figure 4.1* illustrates such a *CHB*.

*CHB*s are extremely interesting in terms of performance, as mentioned in *Section 3.4*. Indeed, a client wishing to access a *CHB* that is present in cache would not need to initiate network communication as it would be formally ensured of the block's validity.

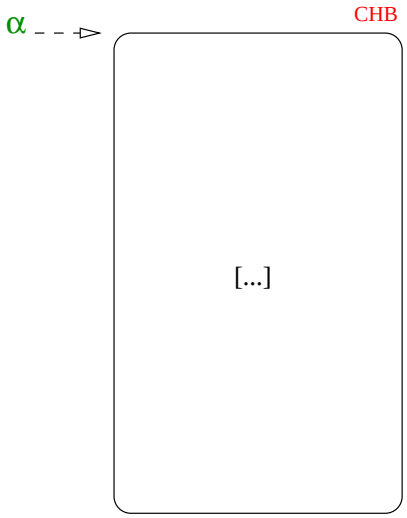


Figure 4.1: The representation of a *CHB*

Algorithms 1, 2 and 3 detail the set-up, seal and validation processes of *CHB*s, respectively. Note that the function $h()$ denotes a one-way function such as *SHA (Secure Hash Algorithm)* for instance.

1. $\alpha \leftarrow h(\beta)$
2. **return** α

Algorithm 1: $\texttt{Setup}_{\overline{CHB}}(\beta) \longrightarrow \alpha$

nothing to do as implicitly sealed

Algorithm 2: $\texttt{Seal}_{\overline{CHB}}(\alpha, \beta)$

1. **if** $\alpha \neq h(\beta)$ **then**
2. **error** *"the address does not match the block"*
3. **end if**

Algorithm 3: $\texttt{Validate}_{\overline{CHB}}(\alpha, \beta)$

Public Key Block

Unlike *CHB*s, *PKB (Public Key Block)*s are associated with a cryptographic key pair such that the address of such blocks is computed by applying a one-way function on the *PKB*'s public key. Since this key does not change over time, *PKB*s are used as mutable blocks. In order to distinguish a block's multiple versions, a version number is embedded. Besides, a cryptographic signature ensures integrity and authenticity, hence preventing anyone but the *PKB*'s owner from updating the block.

Figure 4.2 details the *PKB* internals. The reader can notice that the block's public key is embedded, along with a version number and a signature which ensures integrity and authenticity while respecting ∇_7. As shown through *Algorithms 4, 5 and 6*, everything necessary to the block's verification process is included within the block, hence ensuring symmetry.

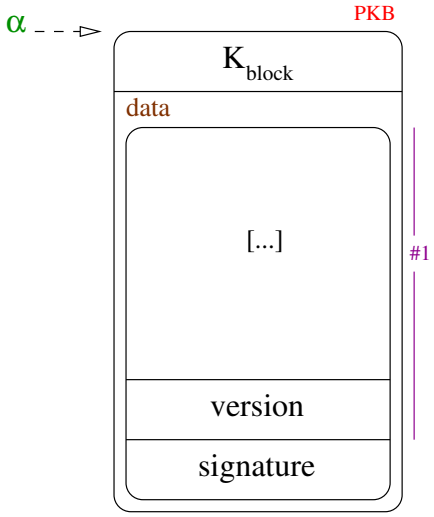


Figure 4.2: The representation of a *PKB*

The reader should carefully consider the notation used throughout this document. While the K and k symbols represent the public and private keys, respectively, $\delta^{\widetilde{\kappa}}$ is equivalent to $\delta^x \ mod \ n$ for $\kappa = (x, n)$ such that:

$$\left(\delta^{\widetilde{K}}\right)^{\widetilde{k}} = \left(\delta^{\widetilde{k}}\right)^{\widetilde{K}} = \delta \tag{4.1}$$

Therefore, $\delta^{\widetilde{K}}$ designates an encryption or signature verification while $\delta^{\widetilde{k}}$ expresses a signature or decryption. In addition, the reader should notice the presence of grouped attributes designated by $\sharp x$ with x a unique number on a given figure. This grouping functionality is used to simplify the algorithms presented throughout this document. For example, while every field is represented on *Figure 4.2*, *Algorithm 6* uses the $\beta.\sharp 1$ notation which is equivalent to $\beta.[...]|\beta.version|\beta.signature$ with $|$ the concatenation operator.

1. $(K_{block}, k_{block}) \leftarrow$ **generate** cryptographic key pair
2. $\beta.K_{block} \leftarrow K_{block}$
3. $\beta.data.version \leftarrow 0$
4. $\alpha \leftarrow h(\beta.K_{block})$
5. **return** α

Algorithm 4: $\texttt{Setup}_{\overline{PKB}}(\beta) \longrightarrow \alpha$

Require: (K_{block}, k_{block}), the block's randomly generated key pair

1. $\beta.data.signature \leftarrow h(\beta.\sharp 1)^{\widetilde{k_{block}}}$

Algorithm 5: $\texttt{Seal}_{\overline{PKB}}(\alpha, \beta)$

1. **if** $\alpha \neq h(\beta.K_{block})$ **then**
2. **error** *"the address does not match the block"*
3. **end if**
4. **if** $\beta.data.signature^{\widetilde{\beta.K_{block}}} \neq h(\beta.\sharp 1)$ **then**
5. **error** *"the data signature is invalid"*
6. **end if**

Algorithm 6: $\texttt{Validate}_{\overline{PKB}}(\alpha, \beta)$

As detailed in *Algorithm 6*, the first step of the validation process verifies that the internal public key $\beta.K_{block}$ is related to the block by applying the one-way function on the public key and comparing the result with the address. Once the public key is known to be valid, the signature can be verified in the second step ensuring the block's integrity and authenticity. From this point on, the block is known to be valid and can therefore safely be used.

Owner Key Block

As described previously, *PKB*s enable users to make use of mutable blocks. However, by relying on such blocks, users will end up keeping as many key pairs as they have created blocks. Unfortunately, ∇_4 stipulates that the access control scheme should not require users to store an increasing amount of access information.

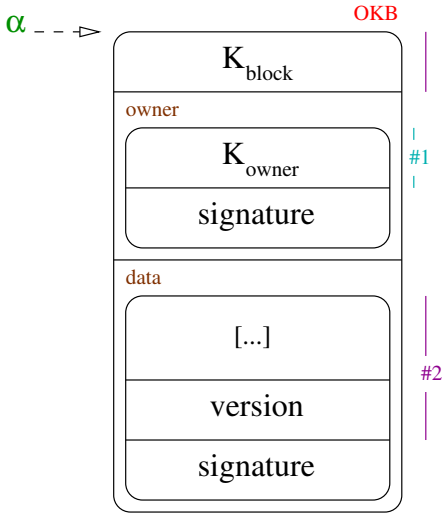


Figure 4.3: The representation of an *OKB*

For this reason, the *OKB (Owner Key Block)* has been introduced, following the original idea from *Pastis* [mBPS05]. Indeed, assuming that every user possesses a single identity key pair, *OKB*s enable users to create blocks without having to keep any access information.

There are two differences between *OKB*s and *PKB*s. Firstly, the owner's public key is recorded in the block and signed with the block's private key. Secondly, the data are no longer signed with the block's private key but with the owner's private key. Thus, since operations are now performed with the owner's key pair, the block's key pair is no longer necessary and can therefore be discarded.

Figure 4.3 depicts the *OKB* internal organisation which shows the inclusion of the block's public key, followed by the owner's public key which is then signed with the block's private key. In addition, *Algorithms 7, 8 and 9* illustrate *OKB*s' set-up, seal and validation processes, respectively.

Require: (K_{user}, k_{user}), the user's personal key pair

1. $(K_{block}, k_{block}) \leftarrow$ **generate** cryptographic key pair
2. $\beta.K_{block} \leftarrow K_{block}$
3. $\beta.owner.K_{owner} \leftarrow K_{user}$
4. $\beta.owner.signature \leftarrow h(\beta.\sharp 1)^{\widetilde{k_{block}}}$
5. $\beta.data.version \leftarrow 0$
6. $\alpha \leftarrow h(\beta.K_{block})$
7. **return** α

Algorithm 7: $\texttt{Setup}_{\overline{OKB}}(\beta) \longrightarrow \alpha$

Require: (K_{user}, k_{user}), the user's personal key pair

1. $\beta.data.signature \leftarrow h(\beta.\sharp 2)^{\widetilde{k_{user}}}$

Algorithm 8: $\texttt{Seal}_{\overline{OKB}}(\alpha, \beta)$

1. **if** $\alpha \neq h(\beta.K_{block})$ **then**
2. **error** *"the address does not match the block"*
3. **end if**
4. **if** $\beta.owner.signature^{\widetilde{\beta.K_{block}}} \neq h(\beta.\sharp 1)$ **then**
5. **error** *"the owner signature is invalid"*
6. **end if**
7. **if** $\beta.data.signature^{\widetilde{\beta.owner.K_{owner}}} \neq h(\beta.\sharp 2)$ **then**
8. **error** *"the data signature is invalid"*
9. **end if**

Algorithm 9: $\texttt{Validate}_{\overline{OKB}}(\alpha, \beta)$

One may have noticed that the data signature is applied on $\beta.K_{block}$—*i.e.* $\beta.\sharp 2$ includes $\beta.K_{block}$—though this was not the case for *PKB*s. The inclusion of $\beta.K_{block}$ in the data signature is necessary to prevent *Injection Attack*s. Indeed, considering two blocks β_1 and β_2 created by the same owner, a malicious user could copy the data section from β_1 and inject it into β_2. This operation would be viewed as a perfectly valid update[1] performed by the owner. Therefore, in order to prevent this kind of attack, the block's public key $\beta.K_{block}$ is included in the data signature, ensuring that the data section is linked to this block, hence cannot be injected in another *OKB*.

4.1.5.2 Logical Blocks

The following presents the logical blocks which introduce concepts such as users, groups *etc.* built on top of the physical blocks detailed above.

User

Because *OKB*s require the notion of user but also because access permissions will eventually be associated with users, this section introduces the *User* block. A *User* block represents a user entity in the storage system and contains information such as the user name, her email address *etc.*

Interestingly, although based on a *PKB*, the *User* block does not require a cryptographic key pair to be generated. Indeed, as mentioned earlier, every user is assumed to possess a unique key pair. Therefore, instead of generating a random key pair for the block to become mutable, the user's personal key pair is used.

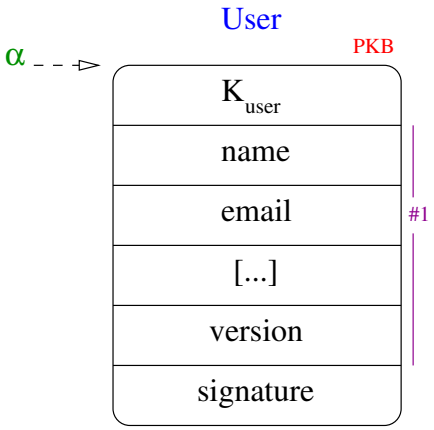


Figure 4.4: The representation of a *PKB*-based *User* block

[1] ... assuming $\beta_1.data.version > \beta_2.data.version$

This characteristic is particularly interesting because, as described before, the address of a *PKB* is computed by applying a one-way function on the block's public key, which happens to be, in this very specific case, the user's public key. Thus, anyone given the user's public key can compute the address of the *User* logical block, which can then be retrieved in order to get additional information on the user.

Figure 4.4 depicts the *User* logical block internals while *Algorithms 10, 11 and 12* detail the set-up, seal and validation processes, respectively, though almost identical to *OKB*'s.

Require: (K_{user}, k_{user}), the user's personal key pair

1. $\beta.K_{user} \leftarrow K_{user}$
2. $\beta.version \leftarrow 0$
3. $\alpha \leftarrow h(\beta.K_{user})$
4. **return** α

Algorithm 10: $\texttt{Setup}_{\frac{User}{PKB}}(\beta) \longrightarrow \alpha$

Require: (K_{user}, k_{user}), the user's personal key pair

1. $\beta.signature \leftarrow h(\beta.\sharp 1)^{\widetilde{k_{user}}}$

Algorithm 11: $\texttt{Seal}_{\frac{User}{PKB}}(\alpha, \beta)$

1. **if** $\alpha \neq h(\beta.K_{user})$ **then**
2. **error** *"the address does not match the block"*
3. **end if**
4. **if** $\beta.signature^{\widetilde{\beta.K_{user}}} \neq h(\beta.\sharp 1)$ **then**
5. **error** *"the signature is invalid"*
6. **end if**

Algorithm 12: $\texttt{Validate}_{\frac{User}{PKB}}(\alpha, \beta)$

Group

A group represents a collection of users and/or sub-groups, administered by a single user, the group's owner, often referred to as the group manager.

In order to fulfill the ∇_6 requirement, the system isolates the group metadata from the members' listing. While the group metadata are recorded in a mutable *Group* logical block, the actual *ACL (Access Control List)* of members is recorded in an immutable *Members* block. This separation has been introduced to minimise the size

of the *Group* block. Indeed, as detailed through *Section 3.4*, retrieving a mutable block such as a *PKB* or an *OKB* requires a quorum of storage nodes to be contacted. Since $2\gamma + 1$ instances of such a mutable block will be transferred back to the requesting client, reducing the size of mutable blocks would drastically increase the system's overall performance. On the other hand, since immutable blocks such as *CHB*s can benefit from caching techniques and, in the worst case scenario, are transferred only once to the client, these blocks can embed far more information without damaging the system's performance.

The *Group* logical block is therefore based on an *OKB* physical block, being modifiable by the group manager, its creator, only. On the other hand, the *Members* logical block is based on a *CHB*. Thus, whenever the list of members is modified, a separate *Members* block is created requiring the *Group* block to be updated in order to reference the new *Members* block.

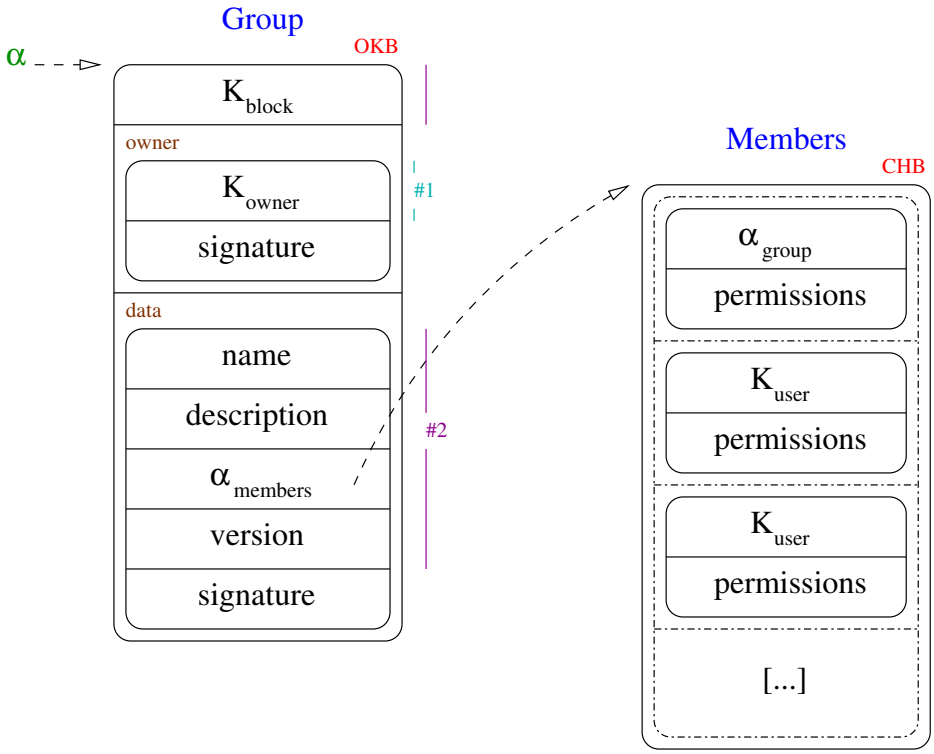


Figure 4.5: The representation of an *OKB*-based *Group* block

Figure 4.5 details the *Group* and *Members* logical blocks and their relation. The reader will notice that the *Members* block contains a list of either user or group entries. While users are identified by their personal public key, groups are referenced by the address of the associated *Group* logical block.

Noteworthy is that permissions are associated to members such that, to be allowed to read an object for instance, a user must identify herself through a chain of group

memberships in which, every group in the chain must have been granted the read permission, including her own user membership. This enables fine-grained group memberships management since a group owner can easily include a sub-group without granting too much permission.

The set-up, seal and validation processes are not detailed for the *Group* and *Members* logical blocks since identical to the *OKB*'s and *CHB*'s.

Object

An *Object* is a protective layer built above actual data blocks which enables users to control access both in reading and writing. Therefore, as its name indicates, it can be used, in the context of file systems, to represent file system objects such as files, directories *etc.*

An object is linked with a set of permissions granted to specific users and/or groups. In order to optimise the most common case in which an object is only accessible to its owner, the owner's permissions are directly recorded in the *Object* block while prospective users and groups are listed in a separate and optional block, the *Access* logical block. Note that, as for the *Group* block, this specific arrangement reduces the size of the *Object* block and therefore optimises the communication costs. *Figure 4.6* illustrates this optimisation, especially in the $\beta.meta$ section where the owner's permissions and the address of the *Access* block are recorded.

As mentioned in *Section 4.1.4*, access beneficiaries are classified as either lords or vassals. Since the fundamental idea behind the access control scheme is not to constrain lords, these more privileged users are given the key for decrypting the data, assuming they have been granted read permission. Such a key comes in the form of a *token* which consists in the key being encrypted with the lord's public key. On the other hand, vassals are not given the key and it is their responsibility to contact a lord whenever they wish to read the data. Once the legitimacy of the operation has been verified by a lord, the key is handed to the vassal so it can decrypt the data.

Regarding write operations, the user updating the *Object* must attach a proof showing evidence of the action's legitimacy. Since lords have been directly granted the permission by the object's owner, providing such a proof comes down to specifying the location, in the *Access* block, of their user entry. Therefore, anyone retrieving the block can verify that the signature has been issued by the private key associated with the public key recorded in the given user entry but also that the permissions associated with this user include the right to update the object. The process is however a bit more complicated when it comes to vassals. Indeed, these users must request a lord to validate their action. The lord then issues a vouching certificate which can be attached to the *Object* block, hence proving the rightfulness of the

modification. *Figure 4.6* illustrates these particularities: $\beta.author.lord$ contains the index of the user entry in the *Access* block while $\beta.author.voucher$ represents the certificate issued by a lord vouching for the user, certificate which includes the public key of the vassal having updated the object. Note that, should the owner update the block, no proof would be provided *i.e.* $\beta.author = \perp$. Likewise, the proof provided by a lord does not include a voucher, hence $\beta.author.voucher = \perp$.

According to ∇_2, a user losing the read permission should no longer be able to read the data. Therefore, the data blocks should be re-encrypted with a new key which would be distributed to the readers. However, since this process is very expensive, most distributed systems delay the re-encryption until the data is modified, a process referred to as *Lazy Revocation* [Fu99]. Unfortunately, in the presented system, a group manager having been granted the read permission on the object could decide to evict a user. As such, users could lose the read permission at any time without the object's owner being aware of such events. Since there is no way for the owner to know when the data must be re-encrypted, a new key is generated every time the object is updated. Note however that this does not mean that all the data blocks are re-encrypted. Indeed, only the modified and new data blocks are encrypted with the new key. In addition, a *Contents* block contains the list of all the data blocks along with the key used for encrypting every one of those blocks. Finally, the *Contents* block is encrypted with the key having been generated for the last writing, the one which is distributed to authorised lords.

Since the owner's connectivity cannot be guaranteed whenever the object is modified, the new key must be generated by the writer. The writer thus generates the tokens based on the new key and distributes them to the read lords. Note that a malicious writer could perform an attack by distributing different—valid or invalid—keys to different lords such that a user retrieving the key would not be able to know which one of the data or the key is incorrect. In order to reduce the risks of such an attack, the writer also attaches a fingerprint of the key. Therefore, readers retrieving a key mismatching the fingerprint would know that the writer is malevolent and complaints regarding this user could then be made to the object's owner. This specificity is illustrated in *Figure 4.6*, especially in the $\beta.data$ section.

The fact that the writer, often referred to as the *author*, updates the lords' tokens implies that she must be allowed to modify the *Access* block. However, this block also embeds the lords' identities and permissions, information that only the owner should be authorised to modify. In order to guarantee that the author can re-generate the lords' tokens, the address of the *Access* block is left under the control of the writer, hence is included in the signature $\beta.data.signature$. However, in order to prevent the writer from modifying permissions, the sensitive data are also included in the signature issued by the owner *i.e.* $\beta.meta.signature$. Therefore, a malicious user trying to illegally modify metadata would inevitably render the block invalid by

∇_1: environment, ∇_2: modification, ∇_3: consultation, ∇_4: space, ∇_5: flexibility, ∇_6: efficiency, ∇_7: accountability

violating the owner's meta signature.

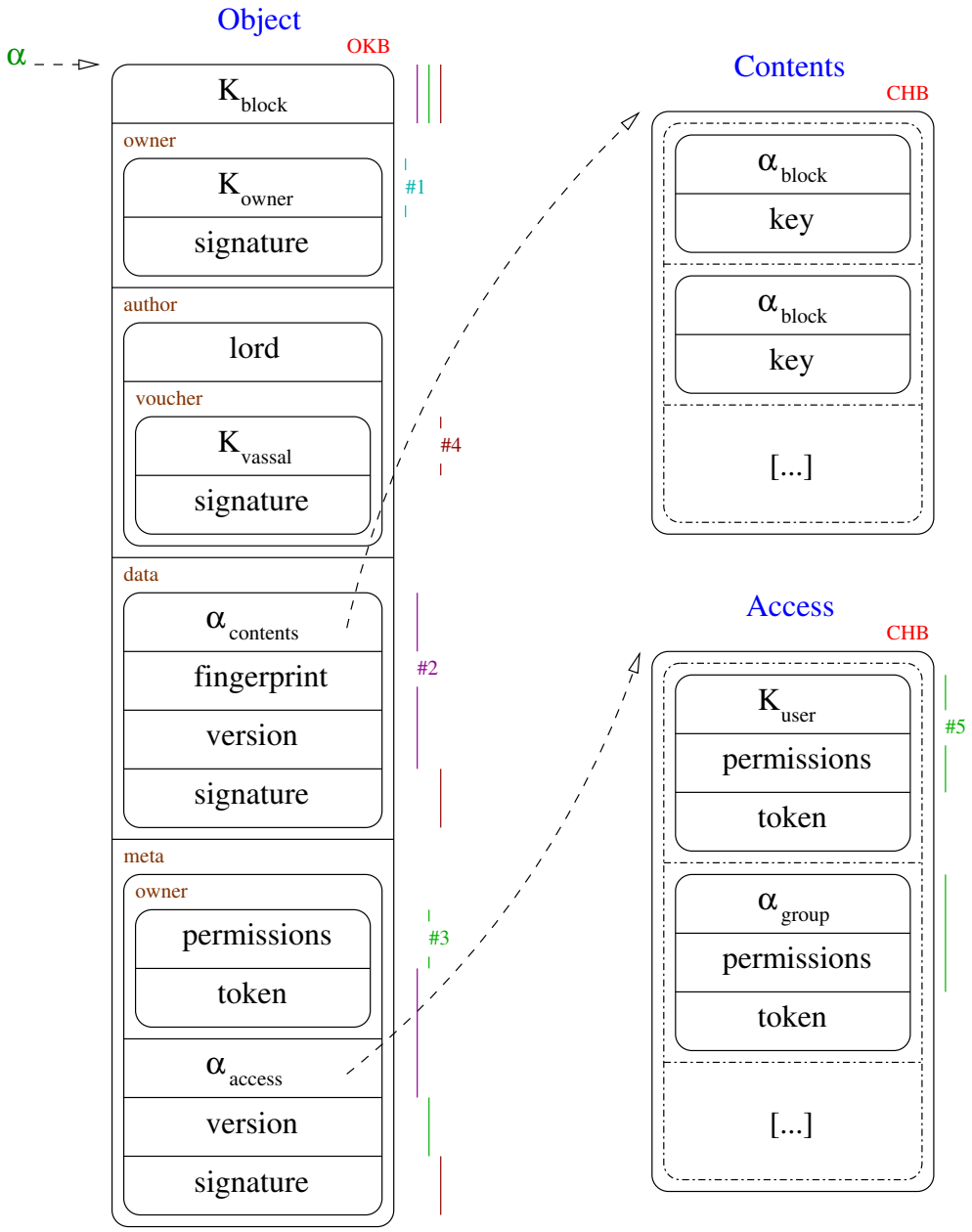


Figure 4.6: The representation of an *OKB*-based *Object* block

Algorithms 13 and 16 detail the set-up and validation processes, respectively. *Algorithm 14* describes the sealing steps for the *Object*'s data section, while *Algorithm 15* describes those for the meta section.

Require: (K_{user}, k_{user}), the user's personal key pair

1. $(K_{block}, k_{block}) \leftarrow$ **generate** cryptographic key pair
2. $\beta.K_{block} \leftarrow K_{block}$
3. $\beta.owner.K_{owner} \leftarrow K_{user}$
4. $\beta.owner.signature \leftarrow h(\beta.\sharp 1)^{\widetilde{k_{block}}}$
5. $\beta.data.version \leftarrow 0$
6. $\beta.meta.owner.permissions \leftarrow \left\{read, write\right\}$
7. $\beta.meta.version \leftarrow 0$
8. $\alpha \leftarrow h(\beta.K_{block})$
9. **return** α

Algorithm 13: $\underset{\frac{Object}{OKB}}{\texttt{Setup}}(\beta) \longrightarrow \alpha$

Require: (K_{user}, k_{user}), the user's personal key pair

1. $\beta.data.signature \leftarrow h(\beta.\sharp 2)^{\widetilde{k_{user}}}$

Algorithm 14: $\underset{\frac{Object[data]}{OKB}}{\texttt{Seal}}(\alpha, \beta)$

Require: (K_{user}, k_{user}), the user's personal key pair

1. **if** $\beta.meta.\alpha_{access} = \perp$ **then**
2. $\quad \hbar \leftarrow h(\beta.\sharp 3)$
3. **else**
4. $\quad \chi \leftarrow \texttt{Get}(\beta.meta.\alpha_{access})$
5. $\quad \hbar \leftarrow h(\beta.\sharp 3 | \chi.\sharp 5)$ – *the | operator designates concatenation*
6. **end if**
7. $\beta.meta.signature \leftarrow \hbar^{\widetilde{k_{user}}}$

Algorithm 15: $\underset{\frac{Object[meta]}{OKB}}{\texttt{Seal}}(\alpha, \beta)$

1. **if** $\alpha \neq h(\beta.K_{block})$ **then**
2. **error** *"the address does not match the block"*
3. **end if**
4. **if** $\beta.owner.signature^{\widetilde{\beta.K_{block}}} \neq h(\beta.\sharp 1)$ **then**
5. **error** *"the owner signature is invalid"*
6. **end if**
7. **if** $\beta.meta.\alpha_{access} = \perp$ **then**
8. $\hbar \leftarrow h(\beta.\sharp 3)$
9. **else**
10. $\chi \leftarrow \texttt{Get}(\beta.meta.\alpha_{access})$
11. $\hbar \leftarrow h(\beta.\sharp 3 | \chi.\sharp 5)$
12. **end if**
13. **if** $\beta.meta.signature^{\widetilde{\beta.owner.K_{owner}}} \neq \hbar$ **then**
14. **error** *"the meta signature is invalid"*
15. **end if**
16. **if** $\beta.author = \perp$ **then**
17. **if** $write \notin \beta.meta.owner.permissions$ **then**
18. **error** *"the owner does not have the permission"*
19. **end if**
20. **if** $\beta.data.signature^{\widetilde{\beta.owner.K_{owner}}} \neq h(\beta.\sharp 2)$ **then**
21. **error** *"the data signature is invalid"*
22. **end if**
23. **else**
24. $\chi \leftarrow \texttt{Get}(\beta.meta.\alpha_{access})$
25. **if** $write \notin \chi[\beta.author.lord].permissions$ **then**
26. **error** *"the lord does not have the permission"*
27. **end if**
28. $K_{lord} \leftarrow \chi[\beta.author.lord].K_{user}$
29. **if** $\beta.author.voucher = \perp$ **then**
30. $K_{author} \leftarrow K_{lord}$
31. **else**
32. **if** $\beta.author.voucher.signature^{\widetilde{K_{lord}}} \neq h(\beta.\sharp 4)$ **then**
33. **error** *"the voucher signature is invalid"*
34. **end if**
35. $K_{author} \leftarrow \beta.author.voucher.K_{vassal}$
36. **end if**
37. **if** $\beta.data.signature^{\widetilde{K_{author}}} \neq h(\beta.\sharp 2)$ **then**
38. **error** *"the data signature is invalid"*
39. **end if**
40. **end if**

Algorithm 16: $\texttt{Validate}_{\frac{Object}{OKB}}(\alpha, \beta)$

The validation process detailed through *Algorithm 16* starts, as for *OKB*s, by verifying that the block corresponds to its associated address. Then, the signature in the owner section is verified proving the owner's public key valid. Thus, the meta signature can be verified guaranteeing that the identities and permissions of the lords have not been altered. Then, depending on the nature of the author—being the owner, a lord or a vassal—, the data signature is verified, leading to the assurance that the tokens are valid along with the encrypted *Contents* block.

Let us consider the following scenario: the owner decides to modify the access control records by removing the write permission from a lord. Unfortunately, this lord happens to be the user having performed the latest modification on the object. As such, the author field references his record in the *Access* block while the data section signature has been computed with his private key and his therefore verified with his public key. Assume that another client later retrieves the object and starts verifying its validity. The verification process would detect that the author does not have write permission and would inevitably conclude that the object was forged, probably by a malicious user. Although the object is now considered invalid by everyone, it was, at the time, legally constructed. In order to overcome this security issue, should the owner remove the write permission from the user who signed the object's data section, the owner would generate a voucher stating that the user's action was legitimate at the time, no matter what his current permissions in the *Access* block are.

Finally, it is important to note that lords' behaviour cannot be guaranteed. Indeed, a malicious lord could, for instance, distribute invalid keys or could even refuse to vouch for a valid vassal. In such a context, a vassal would have no choice but to contact another lord until an honest one is found.

4.1.6 Algorithms

This section provides the reader with a detailed understanding of the algorithms related to three fundamental operations applied to a file system object built upon the blocks described above.

- `Govern`(α, ψ)
- `Read`$(\alpha, \lambda) \longrightarrow \delta$
- `Write`$(\alpha, \lambda, \delta)$

Govern

The `Govern`(α, ψ) routine enables the object's owner to apply a set of modifications ψ on the meta-data, including the access permissions.

This routine starts by retrieving the *Object* block. Then, the meta-data modifications are applied depending on the presence of the *Access* block which may need to be fetched.

Require: (K_{user}, k_{user}), the user's personal key pair

1. $\beta \leftarrow$ `Gather`(α)
2. **if** $K_{user} \neq \beta.owner.K_{owner}$ **then**
3. **error** *"the owner is the only user authorised to govern the object"*
4. **end if**
5. $\beta.meta.version \leftarrow \beta.meta.version + 1$
6. **apply** the set of modifications ψ
7. **for all** (ϵ, ζ) such that ζ is a new immutable block **do**
8. `Seal`$_{\overline{CHB}}(\epsilon, \zeta)$
9. `Put`(ϵ, ζ)
10. **end for**
11. `Seal`$_{\frac{Object[meta]}{OKB}}(\alpha, \beta)$
12. `Put`(α, β)

Algorithm 17: `Govern`(α, ψ)

Read

The `Read`(α, λ) routine takes the address of an *Object* block α along with the location λ of the data to read from this object. The routine returns the data δ.

The algorithm starts by retrieving the *Object* block by calling the `Gather`() method. Then, the key κ used for encrypting the data must be retrieved. This process depends on the nature of the user. Indeed, if the user happens to be the object's owner for instance, the key can be extracted by decrypting the token $\beta.meta.owner.token$. A lord, on the other hand, would have to locate its entry in the *Access* block and proceed to the decryption of the token. Finally, a vassal would need to request a lord to validate the rightfulness of this action before providing the vassal the key.

Once the key κ is retrieved, its fingerprint is verified against the one provided by the author *i.e.* $\beta.data.fingerprint$. Then, the *Contents* block can be fetched and decrypted with κ after which the multiple data blocks are accessible. From this point, the routine can read the data identified by λ.

Noteworthy is that an *Access* user entry lacking the read permission does not necessarily mean that the user cannot access the data. Indeed, the user could still be granted the read permission through one or more group memberships, in which case, through a lord's approval, the user could retrieve the encryption key. Nonetheless and for the sake of clarity, this possibility is ignored in *Algorithm 18*.

Require: (K_{user}, k_{user}), the user's personal key pair

1. $\beta \leftarrow \texttt{Gather}(\alpha)$
2. **if** $K_{user} = \beta.owner.K_{owner}$ **then**
3. **if** $read \notin \beta.meta.owner.permissions$ **then**
4. **error** *"the owner does not have the read permission"*
5. **end if**
6. $\kappa \leftarrow \beta.meta.owner.token^{\widetilde{k_{user}}}$
7. **else**
8. $\chi \leftarrow \texttt{Get}(\beta.meta.\alpha_{access})$
9. **if** $\exists \iota \ : \ K_{user} = \chi[\iota].K_{user} \quad \forall \iota \in \left\{1, ..., |\chi|\right\}$ **then**
10. **if** $read \notin \chi[\iota].permissions$ **then**
11. **error** *"the lord does not have the read permission"*
12. **end if**
13. $\kappa \leftarrow \chi[\iota].token^{\widetilde{k_{user}}}$
14. **else**
15. $\Xi \leftarrow \left\{\chi[\iota].K_{user} \ : \ read \in \chi[\iota].permissions \quad \forall \iota \in \left\{1, ..., |\chi|\right\}\right\}$
16. $\nu \leftarrow \texttt{Locate}(\Xi)$
17. $\kappa \leftarrow$ **request** lord ν for the key by sending the message $\langle \texttt{read}, \alpha, K_{user} \rangle$
18. **end if**
19. **end if**
20. **if** $h(\kappa) \neq \beta.data.fingerprint$ **then**
21. **error** *"the key does not match the fingerprint"*
22. **end if**
23. $\sigma \leftarrow \texttt{Get}(\beta.data.\alpha_{contents})$
24. $\xi \leftarrow$ **decrypt** σ with the key κ
25. $\delta \leftarrow$ **read** data from ξ at location λ
26. **return** δ

Algorithm 18: $\texttt{Read}(\alpha, \lambda) \longrightarrow \delta$

Write

The $\texttt{Write}(\alpha, \lambda, \delta)$ routine takes the address of an object along with some data δ and the location λ of the region of the object's data that should be overwritten.

The algorithm starts by retrieving the *Object* block. Then, depending on the user's relation to the object, being the owner, a lord or a vassal, a proof is constructed and attached to the object *i.e.* $\beta.author$. For instance, a lord constructs a proof by specifying the index of its user entry in the *Access* block while a vassal must contact a lord willing to vouch for her. Once the proof is complete, the user can update the data at λ and encrypt it with a freshly generated key κ, which is then distributed to the read lords *i.e.* the lords with the read permission. Finally, the *Object* is sealed and stored in the distributed hash table along with any additional data block.

Note that, although not depicted in *Algorithm 19*, an author without the read permission would have no choice but to overwrite the existing *Contents* block while one with the permission would be able to modify specific portions of the existing data.

Require: (K_{user}, k_{user}), the user's personal key pair

1. $\beta \leftarrow \texttt{Gather}(\alpha)$
2. $\chi \leftarrow \texttt{Get}(\beta.meta.\alpha_{access})$
3. **if** $K_{user} = \beta.owner.K_{owner}$ **then**
4. **if** $write \notin \beta.meta.owner.permissions$ **then**
5. **error** *"the owner does not have the write permission"*
6. **end if**
7. $\beta.author \leftarrow \perp$
8. **else**
9. **if** $\exists \iota \ : \ K_{user} = \chi[\iota].K_{user} \quad \forall \iota \in \left\{1, ..., |\chi|\right\}$ **then**
10. **if** $write \notin \chi[\iota].permissions$ **then**
11. **error** *"the lord does not have the write permission"*
12. **end if**
13. $\beta.author.lord \leftarrow \iota$
14. $\beta.author.voucher \leftarrow \perp$
15. **else**
16. $\Xi \leftarrow \left\{\chi[\iota].K_{user} \ : \ write \in \chi[\iota].permissions \quad \forall \iota \in \left\{1, ..., |\chi|\right\}\right\}$
17. $\nu \leftarrow \texttt{Locate}(\Xi)$
18. $(\iota, \varphi) \leftarrow$ **request** lord ν for a voucher through $\langle \texttt{write}, \alpha, K_{user}, \delta \rangle$
19. $\beta.author.lord \leftarrow \iota$
20. $\beta.author.voucher \leftarrow \varphi$
21. **end if**
22. **end if**
23. $\beta.data.version \leftarrow \beta.data.version + 1$
24. $\kappa \leftarrow$ **generate** cryptographic symmetric key
25. $\xi \leftarrow$ **write** data δ at location λ
26. $\sigma \leftarrow$ **encrypt** ξ with the key κ
27. $\beta.data.\alpha_{contents} \leftarrow \texttt{Setup}_{\overline{CHB}}(\sigma)$
28. $\texttt{Seal}_{\overline{CHB}}(\beta.data.\alpha_{contents}, \sigma)$
29. $\texttt{Put}(\beta.data.\alpha_{contents}, \sigma)$
30. **for all** $\iota \in \left\{1, ..., |\chi|\right\} \ : \ read \in \chi[\iota].permissions$ **do**
31. $\chi[\iota].token \leftarrow \kappa^{\widetilde{\chi[\iota].K_{user}}}$
32. **end for**
33. $\beta.data.fingerprint \leftarrow h(\kappa)$
34. **for all** (ϵ, ζ) such that ζ is a new immutable block **do**
35. $\texttt{Seal}_{\overline{CHB}}(\epsilon, \zeta)$
36. $\texttt{Put}(\epsilon, \zeta)$
37. **end for**
38. $\texttt{Seal}_{\underset{OKB}{\overline{Object[data]}}}(\alpha, \beta)$
39. $\texttt{Put}(\alpha, \beta)$

Algorithm 19: $\texttt{Write}(\alpha, \lambda, \delta)$

4.1.7 Analysis

Section 4.1.2 claimed unfeasible any access control scheme complying with the defined objectives. As shown, any class of access control required some party to provide temporal or contextual information. Unfortunately, since the environment is assumed to be untrustworthy, neither the clients nor the servers can be trusted to provide such a decisive element of proof. The object's owner, being the only authoritative entity, cannot be relied upon either as no guarantee can be made regarding her connectivity to the system. As a result, the connectivity constraint had to be loosened in order to render the environment suitable for the design of a flexible access control scheme.

As mentioned earlier, most centralised systems rely on a single manager through which every request goes. Such managers thus can control the legitimacy of every client action. Unfortunately, distributed systems cannot rely on a single centralised entity and therefore tend to build and maintain managers in a dynamic way through the use of consensus protocols. However, since known to be expensive, these protocols have been ignored as not considered suitable candidates for an access control system in the given large-scale environment.

The presented access control scheme is innovative in the sense that it makes use of managers without achieving consensus. Indeed, the idea behind the described system is to rely on some specific users to act as managers. Since these users do not communicate, synchronise or achieve consensus, the access control scheme can fairly be claimed to be decentralised, as a client only requires to contact a single manager.

Although neither storage nodes nor clients can be trusted, lords do benefit from a special status. Since lords have certain rights over the object, such as the permission to read and/or write its data, they can be considered as being partially trusted. It is equally fair to assume that lords also have the power to act maliciously, by erasing data for instance or by handing the decryption key to an unauthorised user. The concept lying behind the designed access control scheme is therefore to let lords act as managers, by validating requests related to the permissions they have been personally granted on the object. The most important aspect of this scheme is that it does not weaken the system's security because a malicious write lord already had the power to abuse the system by writing data on behalf of another unauthorised user. Likewise, a malicious read lord could have distributed the key to anyone without the system or the object's owner ever noticing.

The access control scheme detailed throughout this section therefore enables users to protect their objects from ill-disposed users and Byzantine storage servers. Furthermore, the proposed model is flexible and expressive enough so that users can organise their friends, family, acquaintances and so forth, hence easing the access control management. Besides, the design complies with the very strong requirements

defined in *Section 4.1.1*. Thus, every user is required to hold a single personal cryptographic key pair in order to operate on the storage system; accountability is ensured regarding object updates; every access modification issued by the object's owner is taking effect as soon as a write quorum of nodes has agreed on storing the updated object's blocks, as detailed in *Section 3.4*, while every user is able to consult the current permissions associated with any object. Finally, the design complies with the specific environment's properties defined in *Chapter 3*.

Noteworthy is that since the access control scheme makes use of managers, it benefits from most of the advantages of active models. Indeed, as mentioned previously, a vassal wishing to, say, read an object would have to attach a proof showing evidence she had the read permission but also that every group of the chain she claimed to be a member of had been granted the read permission as well. This specificity provides objects' owners and group managers with fine-grained access control functionalities. The scheme could also be embellished through the addition of *black list*s for instance. Such a functionality would provide objects' owners and group managers additional flexibility by specifically listing users and/or groups that should be excluded from the set of granted subjects. Therefore, a group manager could, for example, grant read access to a sub-group but also specify that a subset of this sub-group's members are not to be considered by this authorisation. Note however that these additional functionalities would increase the complexity of the process consisting for a lord to verify that a vassal's claim is legitimate as the black list of every group included in the claim would have to be checked.

Although the presented scheme exhibits many interesting properties, the design decisions especially regarding the connectivity requirement's loosening imply a certain number of trade-offs. Firstly, the access control scheme requires many cryptographic signatures to be issued and verified, which in turn impacts the performance. Indeed, while a single signature needed to be verified in *Plutus* [KRS+03], our design requires objects to embed three cryptographic signatures. Secondly, clients wishing to operate on an object through group memberships must explore the group hierarchy until a chain of memberships is found granting the user the sought for permission. Although techniques such as caching can be used to improve this process, it nonetheless represents an expensive task. Thirdly, the access control design largely relies on the assumption that, given a set of users Ξ, at least one non-Byzantine member of Ξ is connected at all times. Indeed, considering an object and its associated lords and assuming that, at a precise time, none of those lords are connected, vassals would become unable to perform legitimate operations. The following details the probability of such an event occurring.

The probability P_O of a vassal being able to perform a legitimate operation is equal to the probability of at least one non-Byzantine lord being connected among the ρ lords associated with the object; the probability of a lord being disconnected is given

by P_D. Note however that the following assumes the lords to act honestly. Indeed, one should carefully distinguish the percentage of Byzantine nodes in a peer-to-peer network from the percentage of malicious users belonging to a small set of privileged users specifically chosen by the object's owner. While Byzantine nodes misbehave in order to perform unauthorised operations such as making a document diseappear for instance, malicious lords have already been granted permission on the object, hence cannot gain anything more. *Equation 4.2* provides a formula for computing the number ρ of lords required to achieve the desired probability P_O depending on the average lord disconnectivity P_D.

$$\rho = \log_{P_D}(1 - P_O) \tag{4.2}$$

As illustrated by *Figure 4.7*, a probability $P_O = 0.99$ can be achieved by relying on five lords, assuming that users are connected to the network 60% of the time *i.e.* have a probability of being disconnected $P_D = 0.4$. These figures indicate that the presented access control can realistically perform in a production environment. Indeed, as discussed in *Section 4.1.3*, while nodes' connectivity can be assumed to be fairly low due to churn, this access control scheme relies on users' connectivity which is inherently higher.

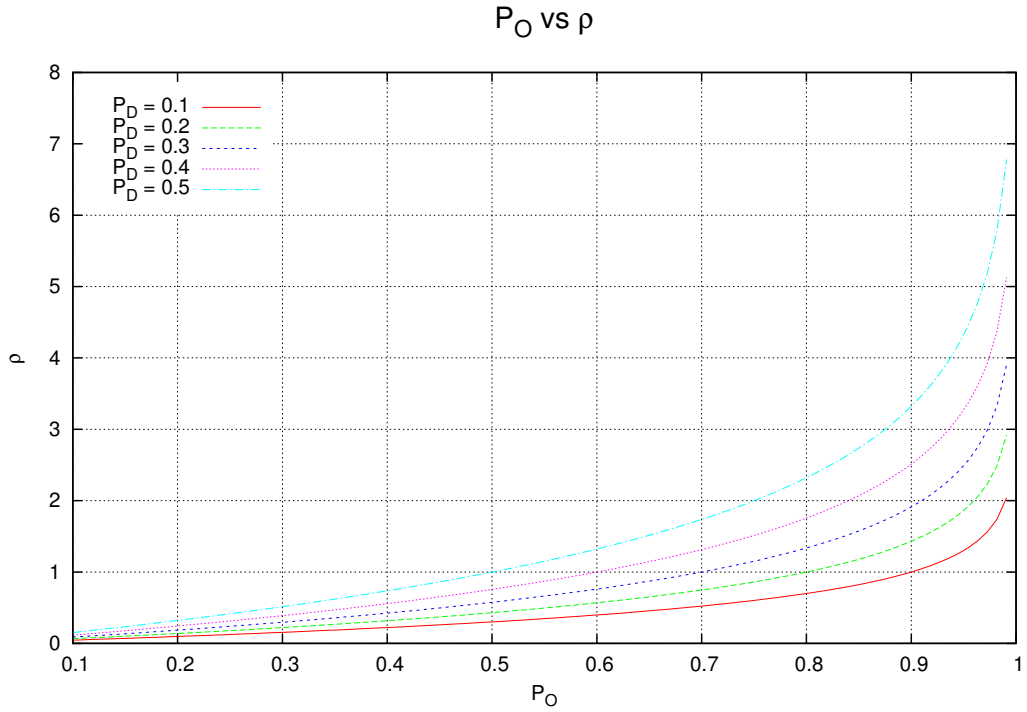


Figure 4.7: A graph showing the relations between P_O and ρ

In addition, a certain number of optimisations can be considered which are described next. Firstly, the connectivity of the group managers can be assumed to be higher

than normal users. Indeed, such users create groups because they want to contribute to the system in more ways than most users. Since some of these group managers also act as lords, access control can benefit from this characteristic. Secondly, the key used for encrypting the *Contents* block could be distributed to vassals in a proactive manner, very much like *Plutus* [KRS+03]. Likewise, lords could try, on a periodic basis, to distribute the key to authorised vassals which were disconnected during the previous attempts. These techniques would increase the probability of a vassal receiving the key, one way or another. Thirdly, vassals having retrieved the key could act as read manager for other vassals. Therefore, a vassal incapable of making contact with a lord could request the key from another vassal. Note that this optimisation is even more interesting when coupled with the proactive distribution while not weakening the overall system's security. Although this optimisation would require users to store a potential large number of keys, such keys could be stored within the system, in a dedicated encrypted block. Nevertheless, vassals would still need to request a lord in order to update the object. However, while delaying read operations, because of lords' unavailability, may degrade the user's experience, delaying write operations is perfectly acceptable and often used to optimise the system's cache, though such delays increase the probability of conflicting updates for objects with multiple writers.

To conclude, the devised access control scheme complies with the very strong requirements imposed by the environment. Indeed, compared to *Plutus* [KRS+03] for instance, this scheme evolves in a completely decentralised and untrustworthy environment, does not make use of expensive algorithms, ensures accountability, guarantees that access modifications are made effective immediately, allows anyone to consult an object's current permissions and requires every user to keep a single item: her personal cryptographic key pair. The major drawback of this system lies in the fact that every object must grant permission to a sufficient number of lords in order to ensure that, at any time, a manager is connected to validate vassals' operations. However, it has been shown that the required connectivity was realistically achievable with a small number of lords, especially through the use of optimisations.

4.2 Administration

Peer-to-peer file systems have emerged in the last decade as a way to provide available and durable storage capacity at low costs by relying on a large-scale set of untrustworthy computers. Some projects such as *CFS* [DKK+01], *Ivy* [MMGC02], *Pastis* [mBPS05], *OceanStore* [KBC+00], *FARSITE* [ABC+02], *Plutus* [KRS+03] and *Chefs* [Fu05] succeeded in offering the user a file system interface. Unfortunately, all of those projects omitted to address one of the fundamental requirement

of such systems, the ability to administer the file system.

In common file systems, being centralised or distributed such as *NFS* [Osa88], *SFS* [MKKW99] *etc,* a user has special privileges allowing her to perform administrative tasks such as creating new users, managing the groups, removing files or directories that she judges inappropriate and so forth. This specific user takes the name of *root* on *Unices* and *Administrator* on *Windows* systems.

Decentralised file systems such as *CFS* [DKK+01], *Ivy* [MMGC02], *Pastis* [mBPS05] *etc.* however, cannot rely on a special and privileged entity because such systems are designed to prevent a single user from taking over the entire system.

The remainder describes the design of an administration scheme that prevents a single user from completely controlling the system while enabling users to request administrative operations.

4.2.1 Semantics

This section discusses the semantics of administrative tasks associated with common file systems and their relevance to the context of decentralised file systems. Note that since the access control scheme presented in *Section 4.1* is based on the *DAC* model, this section focuses on discretionary-access-control-based file systems.

In such systems, an object being a file, directory or link is controlled by a single user, known as the *owner*, in a completely autonomous way. Thus, every operation directed at the object is said to be *object-oriented*. The reader should notice that such operations compose most of a file system's operations.

However, assuming that a file system exposes a unique hierarchical organisation to the users and/or that the data are stored on a hardware device whose access is controlled by the operating system, functionalities operating at the system level become necessary. Since normal users cannot be granted the permission to operate at this critical level, file systems tend to rely on a super-privileged user, named *root* on *Unices* and *Administrator* on *Windows*. Such a *superuser* is commonly allowed to perform the following tasks.

- The superuser can create, delete and update user accounts;
- Likewise, the superuser can create, delete and update group accounts including the list of members and their permissions;
- The superuser is granted all the permissions on the file system. As a consequence, the superuser can create file system objects anywhere in the hierarchical namespace but also access, update and delete any file, directory *etc;* and

- The superuser is also able to change the ownership of a file system object so that object management is transferred to another user.

The following discusses the relevance of such administrative tasks in the context of decentralised file systems.

Entity

As detailed in *Section 4.1*, creating a user in the given peer-to-peer file system comes down to *(i)* generating a cryptographic key pair *(ii)* creating and storing a *User* block. Unlike common file systems in which the superuser manages the user accounts, creating such a block implies that only the owner can update or delete the user entity.

Note however that an organisation wishing to prevent users from creating entities in an autonomous way would have to rely on a *CA (Certification Authority)* for signing the *User*, *Group etc.* blocks associated with the fundamental entities. Although such a *CA* would violate the decentralisation requirement, some research [ACMR02] has been carried out on decentralised certificate management protocols.

It is therefore fair to assume that the creation, update and deletion of user entities can be performed in a completely decentralised way, through the sole management of the *User* block by its owner, without the need for a super-privileged user.

As for users, creating a group in a peer-to-peer file system comes down to storing a *Group* block. Very much like user entities, groups are owned by a single user which is responsible for its management, including maintaing the list of its members along with their permissions.

Therefore, as for users, the creation, update and deletion of group entities can be performed in a completely decentralised way.

Noteworthy is that the notion of user and group often comes with a functionality enabling people to retrieve an entity's identifier given a human-readable representation, such as a name. For instance, *Unices* store the system's users and groups in the files `/etc/passwd` and `/etc/group`, respectively. These files record the human-readable user or group name along with its associated system identifier, known as the *UID (User Identifier)* and *GID (Group Identifier)*. Additionally, *Unices* provide commands such as `id` which takes a username as argument and returns information on the user account including her *UID*, the groups she belongs to *etc.*

Peer-to-peer file systems should integrate similar functionalities, especially because such large-scale systems deal with potentially million of users. Although leaving users the responsibility to communicate their storage user identifier—*i.e.* the user's public key—to their friends might seem reasonable to many systems, such a method would not comply with the transparency requirement defined in *Chapter 3*.

A very simple solution for mapping a username to its identifier would be to create a block whose address is the hash of the username. Therefore, someone looking for a specific user would compute the hash of the username and retrieve this block which would then contain the associated user's public key, hence leading to the *User* block as explained in *Section 4.1.5*. Although such a scheme benefits from an extreme simplicity, it forces users to know the exact names of the entities sought. Thus, one would be unable to look up entities according to a pattern such as a regular expression. Therefore, although functional, this method is limited in terms of expressivity.

Object

As mentioned above, superusers are granted the privilege to access, rename, delete and modify any file system object, independently of its location and without anyone's consent.

The environment's properties defined in *Chapter 3* stipulate that no user should have control over the whole system. This super-privilege therefore seems to strongly conflict with the fundamental properties of the environment. Fortunately, the presented decentralised file system has been designed to prevent any user from accessing an object without the owner's authorisation.

As a consequence, this access super-privilege turns out to be undesirable, unnecessary and unachievable in the given decentralised and discretionary environment.

Noteworthy is that, historically, the notion of a user with super-privileges has been introduced in *UNIX* to deal with the root directory from which the name of the superuser comes. This directory deserves special attention as it is the most critical object in the file system hierarchy. Indeed, common file systems consider that having the right to remove a directory entry is semantically equivalent to being allowed to delete the object the entry points to. Therefore, a user removing a directory entry pointing to a sub-directory indirectly deletes the directory object but also its data, including the sub-entries being files or directories, and so on down to the leaf objects. A file system in which everyone is allowed to modify the root directory would thus inevitably lead to chaos as anyone could destroy branches of the namespace. As a consequence, file systems tend to grant privileges on the root directory to the administrator alone.

The specificity of the peer-to-peer environment implies that such special privileges cannot be granted to a single user. Indeed, and as mentioned earlier, no user should have the power to make file system objects disappear without the owner's consent. However, in order for the file system to evolve, the root directory must be modifiable as every other object in the namespace. Therefore, an administration system adapted to the given environment's characteristics should be provided to overcome

this issue.

Finally, superusers are also granted the right to modify the ownership of an object. Such an operation has probably been provided to the superuser to prevent users from repudiating the ownership by suddenly giving it away to another user without her authorisation or even her awareness.

As detailed in *Section 4.1.5*, the presented peer-to-peer file system, being based on a discretionary access control scheme, makes it impossible for anyone but the owner to modify the object's metadata. Indeed, every *Object* block is associated with a user, known as the object's owner, through a digital signature. This signature is applied on the owner's public key and sealed with the block's private key. Therefore, since the block's private key is known exclusively from the object owner, nobody can modify this object-user link but the owner itself.

Transferring the object's ownership to another user would consist in changing the owner's public key and re-signing it with the block's private key. Unfortunately, this method suffers from two major issues. Firstly, the block's private key has been discarded in order to prevent users from keeping too much access information. Therefore, the signature of the owner's public key could not be re-issued. Secondly, assuming that owners keep the blocks' private key, the owner's public key could indeed be re-signed. However, once the new owner is in place, nothing could prevent the original owner from overwriting the owner's signature once again.

Thus, the file system object representation described throughout *Section 4.1.5* does not seem suitable for enabling the transmission of ownership. An advanced and more specific functionality involving both users to agree on the operation would be necessary.

⋆ ⋆
⋆

In conclusion, most of the privileges granted to a superuser turn out to be unnecessary as such tasks can be performed in a discretionary manner. However, a subset of these operations do not comply with the given environment and therefore require a specific solution. Among those, user and group entities, once created, should be registered in an inventory of some kind. Additionally, the root directory should be accessible by everyone in reading while writing should be carefully supervised in order to prevent rash operations. Finally, an object's owner should be given the possibility to transfer the ownership to another user, assuming the other user agrees to take over.

4.2.2 Model

This section discusses the properties of the *system-wide* and *user-wide* organisation models.

4.2.2.1 System-wide

The vast majority of file systems, being centralised or distributed, expose a single, often hierarchical, namespace to the users such that everyone experiences the same organisation of directories and files. Such file systems are therefore said to make use of a system-wide organisation model because the organisation is applied at the system level, independently of the users' preferences. Note that such a model is being used by a variety of other systems, from the *DNS (Domain Name System)* [Com85, MD88] to *Wikipedia*.

Considering such a model implies numerous obvious advantages but also unexpected issues, especially in the context of peer-to-peer networks. Indeed, as mentioned in *Section 3.4*, peer-to-peer file systems store data blocks in an underlying distributed hash table. The particularity of such file systems compared to centralised ones is that every object is independent of the others. As such, the link existing between a file and the directory that references it is more logical than physical. While a malicious user would be unable to render a centralised file system inconsistent, in a peer-to-peer file system, nothing can prevent a user from destroying a file without updating its parent directory for instance. Should such a scenario occur, users would be able to browse the directory normally, but trying to access the file by fetching the *Object* block would inevitably result in a system failure.

Noteworthy is that an honest user deleting one of her files may end up in the same situation, because not authorised to update the parent directory. Indeed, since the access control model the presented file system relies upon is discretionary, a user must have the permission to destroy the file and its data blocks from the distributed hash table along with the right to update the file's parent directory. Besides, coherency within an object cannot be guaranted either. Indeed, a malicious user legitimately updating a directory *Object* block may deliberately build the directory entries in a way which does not comply with the file system's format. Thus, any user reading this directory would be incapable of understanding its content. Such an attack is obviously inconceivable in a centralised or distributed file system because every request is verified and applied by an authoritative entity.

Therefore, although consistency is naturally expected from file systems, it turns out that such a property is unachievable in file systems relying on a system-wide organisation model and devoid of any authoritative entity for controlling that every update complies with the system's format. Note however that assuming that most

users follow the protocols, consistency should be maintained most of the time and manually fixed when violated.

Finally, since peer-to-peer file systems cannot rely on a superuser, such systems suffer from the issues discussed in *Section 4.2.1* such as the transfer of ownership, the permissions on the root directory *etc.*

4.2.2.2 User-wide

People have different cultures, backgrounds and tastes and therefore have different ways of naming and organising information. For instance, one may organise music according to the genre followed by the band name while another may ignore the genre classification.

The user-wide organisation model decouples the data objects from the organisation objects by enabling users to create their own *view*. A view is composed of organisation objects referencing data objects. In the context of file systems, a view is represented by a hierarchy of directories, every directory being stored in an *Object* block. This way, a user can name and organise the file system content, *i.e.* the data objects, according to its preferences. Note that such a model implies that the organisation objects, *i.e.* the directories, are controlled by their respective owners very much like data objects. Indeed, since both directories and files are stored in *Object* blocks, access to such objects can be restricted or shared. Thus, one user can imagine sharing its view with another user. Besides, the application could enable users to use a view for some parts of the namespace and switch to another view for a specific subset of the hierarchy. For example, one might want to use an official view most of the time but switch to the view provided by *Google* when it comes to the directory `/company/google/` as information is believed to be better organised in this specialised view. This feature is similar to the way stackable file systems [HP94, PPD+95, WDG+06] use union mounts in order to alter the namespace exposed to users according to the context.

In such a user-wide model, views evolve independently from the data. Besides, file objects are no longer attached to a single hierarchy as they were in the system-wide model. Therefore, a user deleting an object would not be able to update the various directories referencing it as nobody can know the views involving this object. Note however that the user may update views she owns and/or has agreed to maintain, assuming she has the proper credentials. Therefore, a user-wide file system must be considered as residing in a perpetual inconsistent state, as nothing can guarantee otherwise.

Nonetheless, it is interesting to notice that the *WWW (World Wide Web)* has been built on the same model in which pages evolved independently of the other pages ref-

erencing them through *hyperlink*s. Therefore, whenever a page is moved or deleted, none of the pages referencing it is updated. Instead the hyperlink becomes irrelevant as pointing to an invalid location, the various references being fixed over time as webmasters notice and correct the problem.

Although this model has performed extremely well for the *WWW*, providing users the liberty to express themselves ouside any control, the inconsistency drawback resulting from the many references pointing to invalid locations may be less suitable to file systems than it has been to the *WWW*. Indeed, by relying on such a model in a file system, users may often end up seeing their accesses failing because of an invalid address. Such a behaviour may irritate the user especially because using the local file system has accustomed end-users to reliability and efficiency.

Noteworthy is that the issues mentioned in *Section 4.2.1* are inherent to the system-wide model and must therefore be reconsidered for the user-wide model. For instance, system-wide models suffer from the fact that the root directory is critical as being the base of the hierarhical organisation. In user-wide models however, multiple root directories exist, one for every view. The access permissions of the view's root directory are directly controlled by the view's owner such that a user disagreeing with the way the view is managed can decide to use another view. Unfortunately, the issues related to the transfer of object ownership as well as the necessity for searching the database of users and groups remain.

⋆ ⋆
⋆

Considering both models, it is highly probable that a user-wide organisation model would perform better in terms of the acceptance and expansion of such a large-scale system very much like the constraint-free hyperlinks enabled users to express themselves through *Web* pages. However, as mentioned above, such a model may not be suitable for the file system context as users are expecting reliability and efficiency and would probably be irritated by the lack of consistency.

Although no model seems to perfectly fit with the required properties defined in *Chapter 3*, it is interesting to note that the issues discussed throughout *Section 4.2.1* are relevant to both models, with the exeption of the root directory. Furthermore, one may notice that both the root directory and the users/groups inventory exhibit an identical flaw: the impossibility to prevent rash modifications should the permission be granted to everyone.

The remainder of this section therefore focuses on providing mechanisms for *(i)* preventing impulsive object modifications *(ii)* transferring object ownership.

4.2.3 Objectives

The objectives regarding the design of an administration scheme are closely related to the access control scheme's defined in *Section 4.1.1*.

∇_1 First, the environment fundamental properties defined in *Chapter 3* must be respected. These include decentralisation, scalability, untrustworthiness and symmetry but also efficiency through quorum protocols, non-connectivity due to churn though this property has been refined in *Section 4.1.3*;

∇_2 The set of users allowed to moderate administrative tasks must be modifiable. Note however that such an operation also constitutes an administrative task as nobody should be allowed to suddenly reform this set without the consent of several other users;

∇_3 As for the users and groups having been granted permission on an object, the set of moderators must be consultable. This is required should a user need to complain for instance;

∇_4 The scheme should not require users to store an excessive amount of information related to administrative tasks; and

∇_7 Finally, moderators should be made accountable for the administrative tasks they approve so that their position can be challenged should they fail to honour their duty for example.

Any administration scheme candidate will be considered unsuitable if violating at least one of these objectives. As in *Section 4.1*, the reader will be able to refer himself to the definitions summary located at the bottom of every page.

4.2.4 Scheme

This section describes the administration scheme which is composed of *(i)* a *community* mechanism enabling users to request administrative tasks and *(ii)* a *ownership* user-to-user protocol providing object owners the possibility to transfer their ownership to another user.

4.2.4.1 Community

The community mechanism consists in the introduction of a physical block known as the *TKB (Table Key Block)*. *TKB*s differ from *OKB*s or even *PKB*s in the sense that such blocks are neither owned nor administered by a single user, the block's owner, but instead by a set of users referred to as the *table* of *knight*s. The idea behind the presented mechanism is to require users wishing to update a block to acquire the approval from a majority of knights.

The scheme has been designed in order to respect the various properties related to the given environment, defined in *Section 4.2.3*. For instance, the table of knights has been introduced for scalability purposes. Indeed, while the number of users populating peer-to-peer networks can be assumed to be high, many of those users may also be considered *dead i.e.* users which will no longer connect to the system. Therefore, designing an administration scheme requiring a user to acquire the approval from the majority of all the users would be impractical but also extremely inefficient. In order to comply with ∇_1, the scheme thus requires an extremely small set of users—*i.e.* compared to the total number of users—to contribute further to the system by moderating a specific object's updates. Noteworthy is that, when coupled with the access control mechanism provided by the *Object* logical block, administrative requests can be limited to some specific users by relying on the permissions field. The ability to precisely control which users are granted the permission to request an operation from the table of knights can help limit the administrative load put on the users acting as knights.

According to the modifications introduced by the community mechanism, the permissions must be refined in order to comply with the semantics of *TKB*s. Indeed, the permissions described so far have been used to grant a user or group either the right to read or write data. The table of knights introduces several subtleties that deserve special attention. Firstly, the write permission is irrelevant to such blocks as any operation involving the block's modification is inherently prohibited; any update must be approved by the knights. Therefore, instead of the permission to write the object's data, users are granted the permission to request an update to the table of knights. Secondly, an *Object* logical block is composed of multiple sections. While the data section could be updated by any user having been granted, directly or indirectly, the write permission, the meta section was administered by the block's owner. However, *TKB*s are devoid of the notion of owner, concept which has been replaced by the table of knights. Similarly, the table of knights itself must be modifiable, as stipulated by ∇_2. Indeed, some knights may wish to leave their position, others may be evicted by the community while users may volunteer to join. Updating the metadata and the table of knights both represent operations that, as any other modification, must be approved by the knights. Therefore, the

∇_1: environment, ∇_2: modification, ∇_3: consultation, ∇_4: space, ∇_5: flexibility, ∇_6: efficiency, ∇_7: accountability

permissions field should reflect these extensions through the addition of the *govern*—the administration of the object's metadata—and *elect*—the administration of the table's composition—permissions. Thirdly, the *TKB* physical block could be used by other logical blocks such as *Group*s for instance. Such a construct would enable a set of users to cooperatively manage a group, a characteristic which should be very much appreciated by communities. Unfortunately, no permission has been associated with such operations because the group owner was the only user allowed to administrate the *Group* block. Additional permissions should thus be introduced in order to provide group members the right to request the modification of the *Group*'s table of knights but also the list of members. Since group members could theoretically be indirectly granted the permission to request the modification of the metadata and table of knights associated with an *Object*, permissions to update the *Object* and the *Group* blocks should be made distinguishable. Therefore, a group member could be granted the *manage* permission to modify the *Group*'s table of knights and the *edit* permission to modify the *Group*'s data *i.e.* the members and their permissions. Note that the sub-groups recorded in the *Members* block could also be granted those permissions. In addition to these permissions, every group member can be indirectly granted the elect, govern, read and/or write permissions on an *Object*. *Table 4.1* summarises the permissions which can be granted to users according to their role, *i.e.* lord or vassal, in the context of *TKB*s. These extensions demonstrate the flexibility and adaptability of the access control scheme designed in *Section 4.1*.

Figure 4.8 depicts the internal organisation of a *TKB* physical block. One might notice that unlike most of the physical blocks presented in *Section 4.1.5*, *TKB*s embed a *seed*. While the address of mutable blocks such as *PKB*s and *OKB*s is computed by applying a one-way function on the public key of the block's key pair, the address of a *TKB* is computed by applying a one-way function on a randomly generated integer: $\beta.seed$. Indeed, since *TKB*s are devoid of any owner, there is no need to generate a cryptographic key pair. Additionally, the table of knights is included in the block, hence complying with ∇_3. Note that, as for *PKB*s and *OKB*s, the data section contains a version number as well as a signature issued by the author whose public key is also included in the block: $\beta.K_{author}$. Finally, and in order to comply with ∇_7, every acquired knight's vote is attached to the block so that anyone can verify the block's validity: *(i)* every vote is unique and relates to this block *(ii)* a majority of votes has been reached. Noteworthy is that every vote contains an index to the related knight in $\beta.table.board$ along with a signature which has been applied on *(i)* the block's identity *i.e.* the seed *(ii)* the table *(iii)* the author's public key and *(iv)* the data signature. Such inclusions prevent votes from being forged or re-used in other contexts *i.e.* for another block, another operation *etc.*

∇_1: environment, ∇_2: modification, ∇_3: consultation, ∇_4: space, ∇_5: flexibility, ∇_6: efficiency, ∇_7: accountability

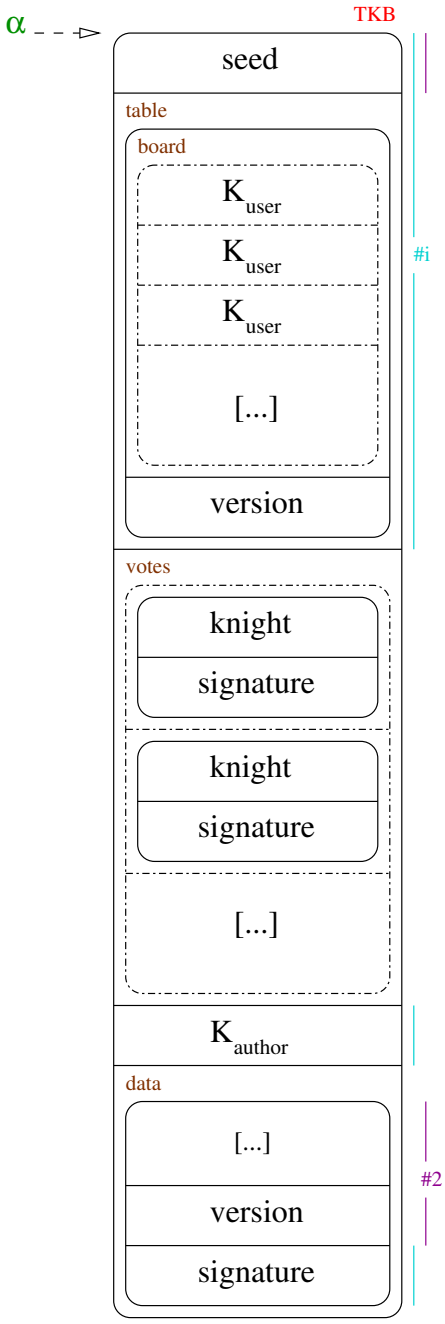


Figure 4.8: The representation of a *TKB*

Subject	Permission	Description
Lord	elect	request a modification of the *Object*'s table of knights
	govern	request a modification of the *Object*'s metadata
	read	read the *Object*'s data
	write	request a modification of the *Object*'s data
Vassal	elect	request a modification of the *Object*'s table of knights
	govern	request a modification of the *Object*'s metadata
	read	read the *Object*'s data
	write	request a modification of the *Object*'s data
	manage	request a modification of the *Group*'s table of knights
	edit	request a modification of the *Group*'s composition

Table 4.1: A summary of the permissions in the file system

Algorithms 20, 21 and 22 detail the set-up, seal and validation processes of *TKB*s, respectively.

Require: (K_{user}, k_{user}), the user's personal key pair

1. $\beta.seed \leftarrow$ **generate** integer
2. $\beta.table.board \leftarrow \left\{K_{user}\right\}$
3. $\beta.table.version \leftarrow 0$
4. $\beta.data.version \leftarrow 0$
5. $\alpha \leftarrow h(\beta.seed)$
6. **return** α

Algorithm 20: $\underline{\texttt{Setup}}_{\overline{TKB}}(\beta) \longrightarrow \alpha$

Require: (K_{user}, k_{user}), the user's personal key pair

1. $\beta.data.signature \leftarrow h(\beta.\sharp 2)^{\widetilde{k_{user}}}$

Algorithm 21: $\underline{\texttt{Seal}}_{\overline{TKB}}(\alpha, \beta)$

1. **if** $\alpha \neq h(\beta.seed)$ **then**
2. **error** *"the address does not match the block"*
3. **end if**
4. **if** $|\beta.votes| < \left\lceil \frac{|\beta.table.board|}{2} \right\rceil$ **then**
5. **error** *"the table's majority has not been acquired"*
6. **end if**
7. **for all** $\iota \in \left\{1, ..., |\beta.votes|\right\}$ **do**
8. $K_{knight} \leftarrow \beta.table.board[\beta.votes[\iota].knight]$
9. **if** $\beta.votes[\iota].signature^{\widetilde{K_{knight}}} \neq h(\beta.\sharp i)$ **then**
10. **error** *"the vote signature is invalid"*
11. **end if**
12. **end for**
13. **if** $\beta.data.signature^{\widetilde{\beta.K_{author}}} \neq h(\beta.\sharp 2)$ **then**
14. **error** *"the data signature is invalid"*
15. **end if**

Algorithm 22: `Validate`$[client]_{\overline{TKB}}(\alpha, \beta)$ — client side

One may have noticed that, unlike mutable blocks such as *PKB*s and *OKB*s, *TKB*s contain a section, the table, which is not protected through the use of a cryptographic signature. Indeed, unlike other mutable blocks which are administered by a single and static authority *i.e.* the block's owner, *TKB*s' authority is represented by the table of knights whose composition evolves over time. Since no static relation exists between the block's address and its table of knights, the self-certification property is violated. As a consequence, the table's integrity and authenticity cannot be guaranteed, implying that anyone can theoretically modify the table's composition. Thus, a client could submit a version of the block in which the table's composition has been replaced. The client would transfer the block to a write quorum of storage nodes which would verify the block's validity. According to the table of knights embedded in the block, the votes prove the knights' approval since *(i)* every vote is valid and *(ii)* a majority of votes has been provided. Storage nodes, considering the block as legitimate, would therefore agree on storing the data, hence overwriting the previous version.

Let us consider a user updating an object from version ν_i to ν_{i+1}. In order for the modification to be considered valid, the user has to attach votes showing evidence that the knights at ν_i approved this modification. However, since the operation relates to the modification of the data, the table is not, in theory, being modified so that the tables at ν_i and ν_{i+1} are identitical. Therefore, the votes issued by the knights at ν_i and attached to the block ν_{i+1} can legitimately be checked against the table of knights at ν_{i+1}. Although the semantics regarding data modifications conform to the administration scheme's objectives, the operation consisting in up-

dating the table's composition violates ∇_2, as illustrated by the next scenario. A user wanting to update the table of knights' composition must provide evidence that the current table of knights at ν_i approves the future composition which will be attached to the instance ν_{i+1}. Unfortunately, the validation process, detailed in *Algorithm 22*, checks the votes against the embedded table of knights *i.e.* both at ν_{i+1}. The validation process can therefore be considered as fundamentally flawed when it comes to verifying a block whose table of knights has been modified.

Besides, should the `Validate`$[client]_{\overline{TKB}}(\alpha, \beta)$ routine be revised to check every vote in ν_{i+1} against the table of knights at ν_i, the symmetry property[2] would be violated. Indeed, since clients cannot have access to the current and past versions of a given block, this verification step could not be performed on the client-side. Thus, as detailed in *Section 3.4*, clients would have no choice but to trust the servers, hence inevitably violating the untrustworthiness property.

Although the system has been designed around quorum algorithms because such algorithms exhibit better performances than their agreement counterpart, such algorithms appear impotent regarding this issue. Indeed, quorum algorithms have shown to perform well in the given environment because the validation process could detect any illegitimate block instance. As such, by acquiring a read quorum composed of $2\gamma + 1$ instances, the potential γ invalid blocks can be identified and thus discarded, leaving the client with $\gamma + 1$ valid blocks among which at least one instance is the latest version *i.e.* with the highest version number. However, it has been previously shown that, since the table of knights cannot be statically protected through self-certification, valid instances of *TKB*s cannot be distinguished from illegitimate ones. The storage algorithms must therefore be reconsidered in order to rely on consensus, the only paradigm ensuring the client to retrieve the latest valid non-self-certified instance in a Byzantine environment.

The system's protocols must be modified for the specific purpose of *TKB*s to either *(i)* rely on agreement algorithms such as *BFT* [CL99], *Paxos* [Lam98] *etc.* or *(ii)* rely on specialised quorum algorithms by breaking the symmetry property, as explained in *Section 3.4*. On the one hand, agreement protocols would provide the flexibility required to handle advanced functionalities in which case the set of storage nodes Ω would run a Byzantine protocol ensuring that every client's request is processed by the servers until a consensus is reached. On the other hand, relying exclusively on quorum algorithms would avoid developers having to maintain the source code for both agreement and quorum algorithms. For the sake of coherency within the system but also because agreement protocols are known to be expensive, quorum algorithms will be specialised for the purpose of *TKB*s. Another argument in favour of quorum

[2] ... peer-to-peer nodes are considered equally unprivileged so that everything performed by one node could also be performed by another one

algorithms is that symmetry has never been completely respected. Indeed, although it has never been discussed in detail, every storage node receiving a mutable block to store verifies the block's validity but also checks that the embedded version numbers are increasing in a strict monotonic way. Obviously, this validation step can only be performed on the server-side since, unlike clients, servers can access both the current and future versions of a block. Note however that although this additional verification step violates the symmetry property, it is not fundamentally required for ensuring the system's safety. Indeed, assuming that version numbers are not verified, block versions could increase in a non-monotonic way but this would not prevent clients from retrieving the version with the highest number. Similarly, clients could submit blocks with version numbers being lower than the current ones. Such scenarios are especially likely to occur whenever different clients concurrently update the same block. Therefore a client updating a block from version ν_i to ν_{i+1} could see her operation rejected because the block has been concurrently updated, say thrice, to version ν_{i+3} before receiving the client's update. Without such a version verification, a client could believe that her update has been applied while, in fact, it does not have the highest version number. The submitted instance will therefore never be used leading to the loss of the modifications.

Let us consider another scenario in which the storage nodes do break the symmetry property by verifying the version ν_{i+1}'s votes against the current table of knights *i.e.* which is located in version ν_i. Let us assume that the client wishes the addition of five knights to the block's table for a total of ten knights. The client, by requesting the current knights, starts by acquiring a majority of three votes. The votes are then included in the new block which embeds the future table composed of ten knights. The storage nodes receiving this block verify that *(i)* a majority of votes has been reached according to the table of knights at ν_i—*i.e.* three votes out of five—and *(ii)* the embedded votes have been issued by the knights of this same table. Since, in this scenario, both conditions have been met, storage nodes consider the block as valid and therefore accept it. Let us recall that, as shown in *Algorithm 22*, the client's verification procedure checks the attached votes against the embedded table of knights. Therefore, a client fetching and verifying the block ν_{i+1} would reject it because the three votes attached to the block have been issued by the knights at ν_i and therefore do not match the embedded table which is composed of ten knights. The votes provided for modifying the table of knights should therefore be distinguished from the embedded votes.

The protocol for updating *TKB*s must therefore be slightly improved as detailed next. Every client wanting to update a *TKB*, being the data or the table of knights, must acquire a majority of votes from the knights at ν_i along with a majority from the future knights *i.e.* at ν_{i+1}. Then, the client builds the new block by including the new table along with the votes issued by these future knights. Finally, the client

sends to the storage nodes both the votes issued by the current knights and the new block. Every storage node receiving the block starts by checking the additional votes against the table composition at ν_i—hence proving that the modification has been approved by the current knights—before verifying the block's validity: *(i)* a majority of votes has been reached and *(*ii) the attached votes have been issued by the knights referenced in the block's table.

Algorithm 23 illustrates the validation process from the server's perspective. Note that the verifications regarding the monotonically increasing version numbers are ignored for the sake of simplicity as it has been throughout this chapter. This verification procedure is composed of two steps. First, the additional votes are checked against the table of knights referenced in the current version of the block, hence proving that the current knights approved the modification. Then, the given block is validated by following the client verification procedure: the votes are checked against the embedded table of knights and the data signature is finally verified.

Require: ϑ, the current version of the α block
ε, the set of votes provided by the client and issued by the knights in ϑ

1. **if** $|\varepsilon| < \left\lceil \frac{|\vartheta.table.board|}{2} \right\rceil$ **then**
2. **error** *"the table's majority has not been acquired"*
3. **end if**
4. **for all** $\iota \in \left\{1, ..., |\varepsilon|\right\}$ **do**
5. $K_{knight} \leftarrow \vartheta.table.board[\varepsilon[\iota].knight]$
6. **if** $\varepsilon[\iota].signature^{\widetilde{K_{knight}}} \neq h(\beta.\sharp i)$ **then**
7. **error** *"the vote signature is invalid"*
8. **end if**
9. **end for**
10. **if** $\alpha \neq h(\beta.seed)$ **then**
11. **error** *"the address does not match the block"*
12. **end if**
13. **if** $|\beta.votes| < \left\lceil \frac{|\beta.table.board|}{2} \right\rceil$ **then**
14. **error** *"the table's majority has not been acquired"*
15. **end if**
16. **for all** $\iota \in \left\{1, ..., |\beta.votes|\right\}$ **do**
17. $K_{knight} \leftarrow \beta.table.board[\beta.votes[\iota].knight]$
18. **if** $\beta.votes[\iota].signature^{\widetilde{K_{knight}}} \neq h(\beta.\sharp i)$ **then**
19. **error** *"the vote signature is invalid"*
20. **end if**
21. **end for**
22. **if** $\beta.data.signature^{\beta.\widetilde{K_{author}}} \neq h(\beta.\sharp 2)$ **then**
23. **error** *"the data signature is invalid"*
24. **end if**

Algorithm 23: `Validate`$[server]_{\overline{TKB}}(\alpha, \beta)$ — server side

Since *TKB*s do not comply with the self-certification property, the quorums must be adapted so as to behave in a consensus way. Indeed, rather than relying on the blocks' self-certification property, the *TKB*-specific quorum algorithm relies on the fact that up to γ replicas can be illegitimate such that the valid and latest version can be identified by gathering at least $\gamma + 1$ identical instances. The quorums are thus redefined in order to reflect this paradigm. Firstly, a client wishing to retrieve the block would need to acquire a read quorum composed of $2\gamma + 1$ so that it is guaranteed to identify $\gamma + 1$ identical instances. Secondly, in order to ensure that every storage node provides the block's latest version, a modification requires the client to acquire a write quorum composed of $|\Omega| = 3\gamma + 1$ nodes.

Figure 4.9 depicts a scenario in which two Byzantine nodes collude in order to

mislead the clients. At first, the storage nodes are consistent, each one storing version 4 except for both Byzantine nodes which answer every request with a forged block embedding an illegal table of knights granting them full privileges. Then, after having received the authorisation from the table of knights, a client submits an update by acquiring a write quorum composed of the seven nodes, though the Byzantine nodes ignore the update. Finally, another client acquires a read quorum, receiving three version 5 and two illegal version 9. Following the protocol, the client isolates the $\gamma + 1$ identical instances which happen to be the block's latest and legitimate version 5.

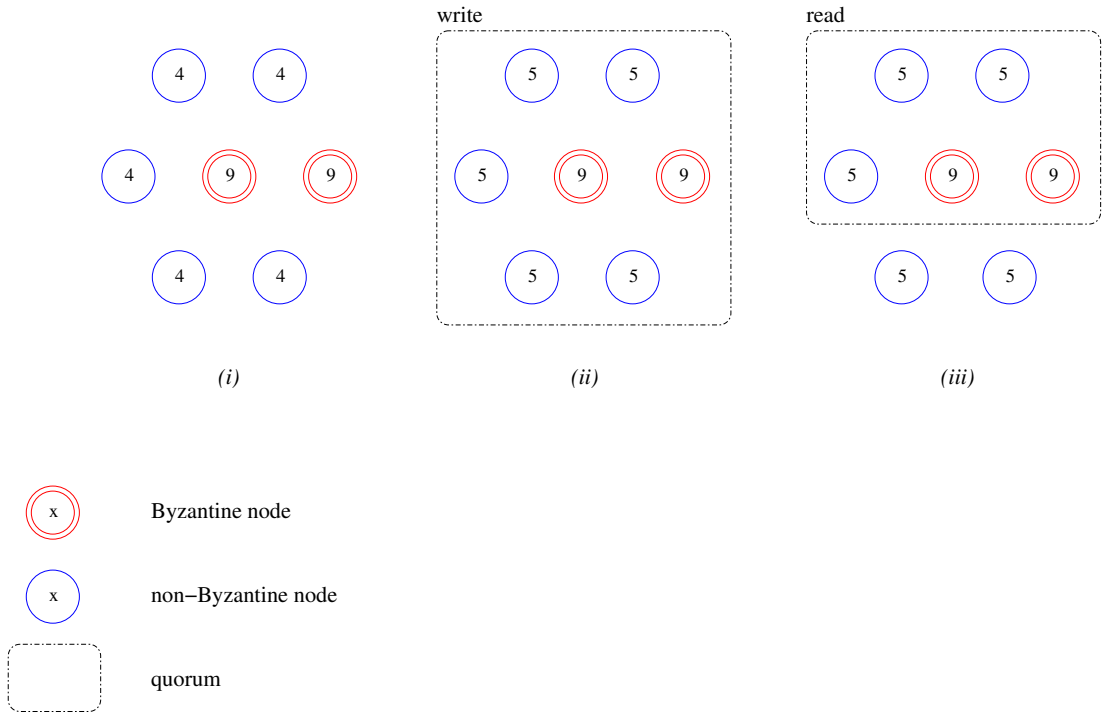


Figure 4.9: A scenario illustrating the *TKB*-specific quorum algorithm

Figure 4.10 illustrates the internal organisation of an *Object* logical block built upon a *TKB*. The *Object* benefits from the community features provided by the *TKB* such that users and groups can be granted the permission to read the data; request a data modification *i.e.* write; request a metadata modification *i.e.* govern or request the modification of the table of knights *i.e.* elect. Apart from these permission extensions, the *Object* block behaves as expected except that, obviously, every request requires the user to acquire the table of knights' consent through the voting process.

The internal organisation differences between *OKB*-based and *TKB*-based *Object* blocks are twofold. First, let us recall that, within the context of *OKB*-based *Objects*, a vassal wishing to update the block needs to request a lord to certify that the vassal had been granted the proper permission. In addition, *TKB*s require users updating the block to request knights to certify that the modification conforms to the

community's interests. Thus, *TKB*-based *Object* blocks theoretically require users to request both a lord and a majority of knights. Since the critical component of this process lies in users' connectivity, requiring users to contact a lord would impact the performance since knights, as the ultimate authority, could perform both certifications. The *TKB*-based *Object*'s internal structure therefore no longer embeds an author section containing the index of the lord along with a potential voucher. Instead, the user updating the data includes her public key in $\beta.K_{author}$ while knights verify that every user requesting an operation has been granted, directly or indirectly, the proper permissions. Second, since *TKB*s are devoid of the notion of owner, the *TKB*-based *Object*'s metadata can be updated by any user having been granted the govern permission. Thus, as for data modifications, a user wanting to update the metadata, including the access permissions, must acquire the authorisation from the table of knights. In addition, assuming the table of knights approved the update, the user must record her public key in the field $\beta.K_{governor}$.

Noteworthy is that the entries of the table of knights do not contain a permissions field. Therefore, in order to be able to perform operations as other users, users acting as knights must also be recorded in the *Access* block, directly as lords or indirectly through group memberships as vassals. Interestingly and quite ironically, knights may not be granted the read permission though they are requested to approve every data modification.

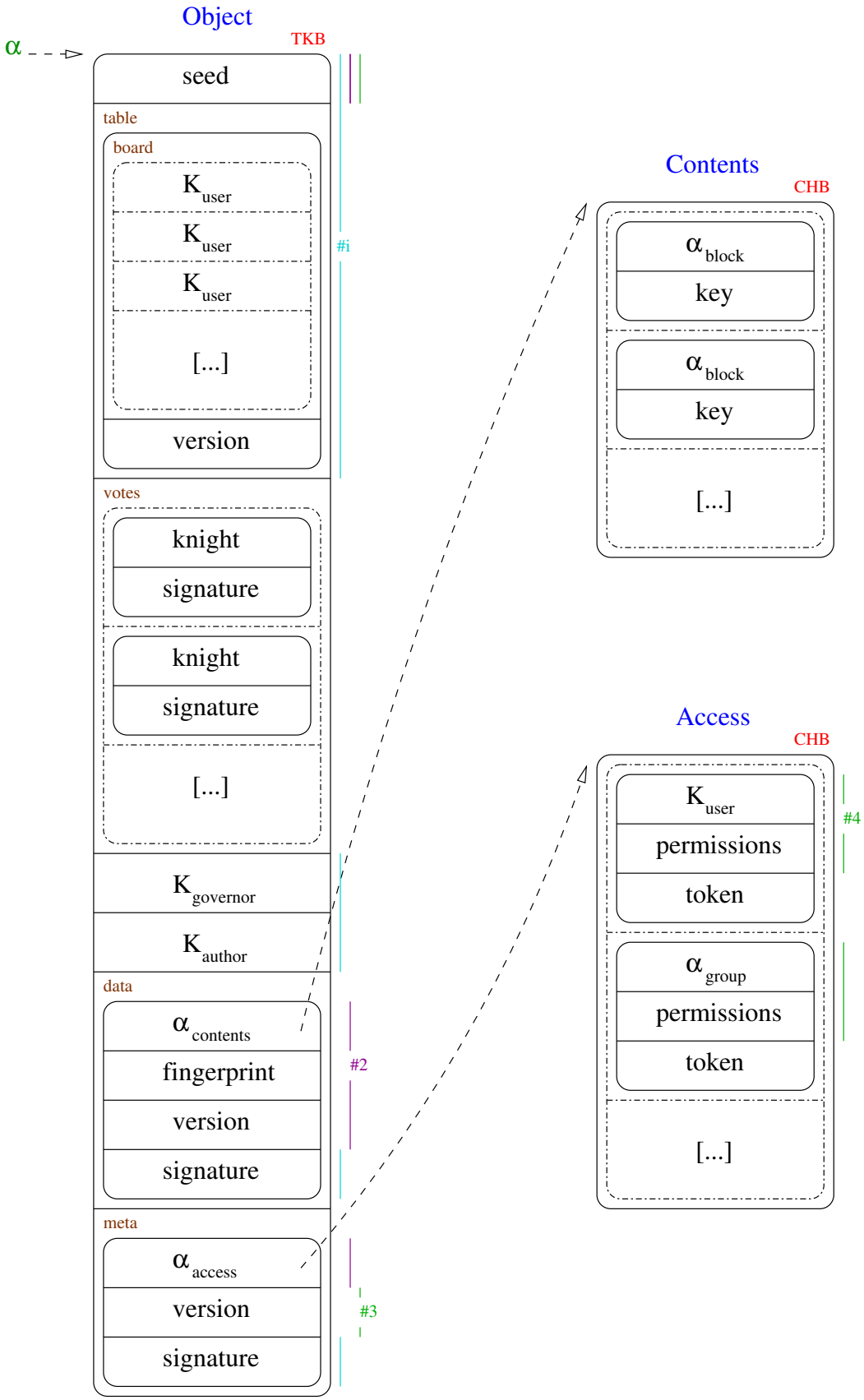


Figure 4.10: The representation of a *TKB*-based *Object* block

Algorithm 24 details the set-up process while *Algorithm 27* describes the client-side validation processes. Note that the server-side validation process is not provided as a combination of *Algorithm 23* and *Algorithm 27*. Also, *Algorithm 25* lists the sealing steps for the *TKB*-based *Object*'s data section while *Algorithm 26* details the meta section's.

Require: (K_{user}, k_{user}), the user's personal key pair

1. $\beta.seed \leftarrow$ **generate** integer
2. $\beta.table.board \leftarrow \left\{K_{user}\right\}$
3. $\beta.table.version \leftarrow 0$
4. $\beta.data.version \leftarrow 0$
5. $\beta.meta.version \leftarrow 0$
6. $\alpha \leftarrow h(\beta.seed)$
7. **return** α

Algorithm 24: $\texttt{Setup}_{\frac{Object}{TKB}}(\beta) \longrightarrow \alpha$

Require: (K_{user}, k_{user}), the user's personal key pair

1. $\beta.data.signature \leftarrow h(\beta.\sharp 2)^{\widetilde{k_{user}}}$

Algorithm 25: $\texttt{Seal}_{\frac{Object[data]}{TKB}}(\alpha, \beta)$

Require: (K_{user}, k_{user}), the user's personal key pair

1. **if** $\beta.meta.\alpha_{access} = \perp$ **then**
2. $\quad \hbar \leftarrow h(\beta.\sharp 3)$
3. **else**
4. $\quad \chi \leftarrow \texttt{Get}(\beta.meta.\alpha_{access})$
5. $\quad \hbar \leftarrow h(\beta.\sharp 3 | \chi.\sharp 4)$
6. **end if**
7. $\beta.meta.signature \leftarrow \hbar^{\widetilde{k_{user}}}$

Algorithm 26: $\texttt{Seal}_{\frac{Object[meta]}{TKB}}(\alpha, \beta)$

1. **if** $\alpha \neq h(\beta.seed)$ **then**
2. **error** *"the address does not match the block"*
3. **end if**
4. **if** $|\beta.votes| < \left\lceil \frac{|\beta.table.board|}{2} \right\rceil$ **then**
5. **error** *"the table's majority has not been acquired"*
6. **end if**
7. **for all** $\iota \in \left\{1, ..., |\beta.votes|\right\}$ **do**
8. $K_{knight} \leftarrow \beta.table.board[\beta.votes[\iota].knight]$
9. **if** $\beta.votes[\iota].signature^{\widetilde{K_{knight}}} \neq h(\beta.\sharp i)$ **then**
10. **error** *"the vote signature is invalid"*
11. **end if**
12. **end for**
13. **if** $\beta.data.signature^{\widetilde{\beta.K_{author}}} \neq h(\beta.\sharp 2)$ **then**
14. **error** *"the data signature is invalid"*
15. **end if**
16. **if** $\beta.meta.\alpha_{access} = \perp$ **then**
17. $\hbar \leftarrow h(\beta.\sharp 3)$
18. **else**
19. $\chi \leftarrow \texttt{Get}(\beta.meta.\alpha_{access})$
20. $\hbar \leftarrow h(\beta.\sharp 3 | \chi.\sharp 4)$
21. **end if**
22. **if** $\beta.meta.signature^{\widetilde{\beta.K_{governor}}} \neq \hbar$ **then**
23. **error** *"the meta signature is invalid"*
24. **end if**

Algorithm 27: $\texttt{Validate}[client]^{\frac{Object}{TKB}}(\alpha, \beta)$ — client side

Finally, *Figure 4.11* details the organisation of a *TKB*-based *Group* block demonstrating how *TKB*s can be adapted to a variety of logical blocks. The set-up, seal and validation processes are not provided as identical to the *TKB*'s.

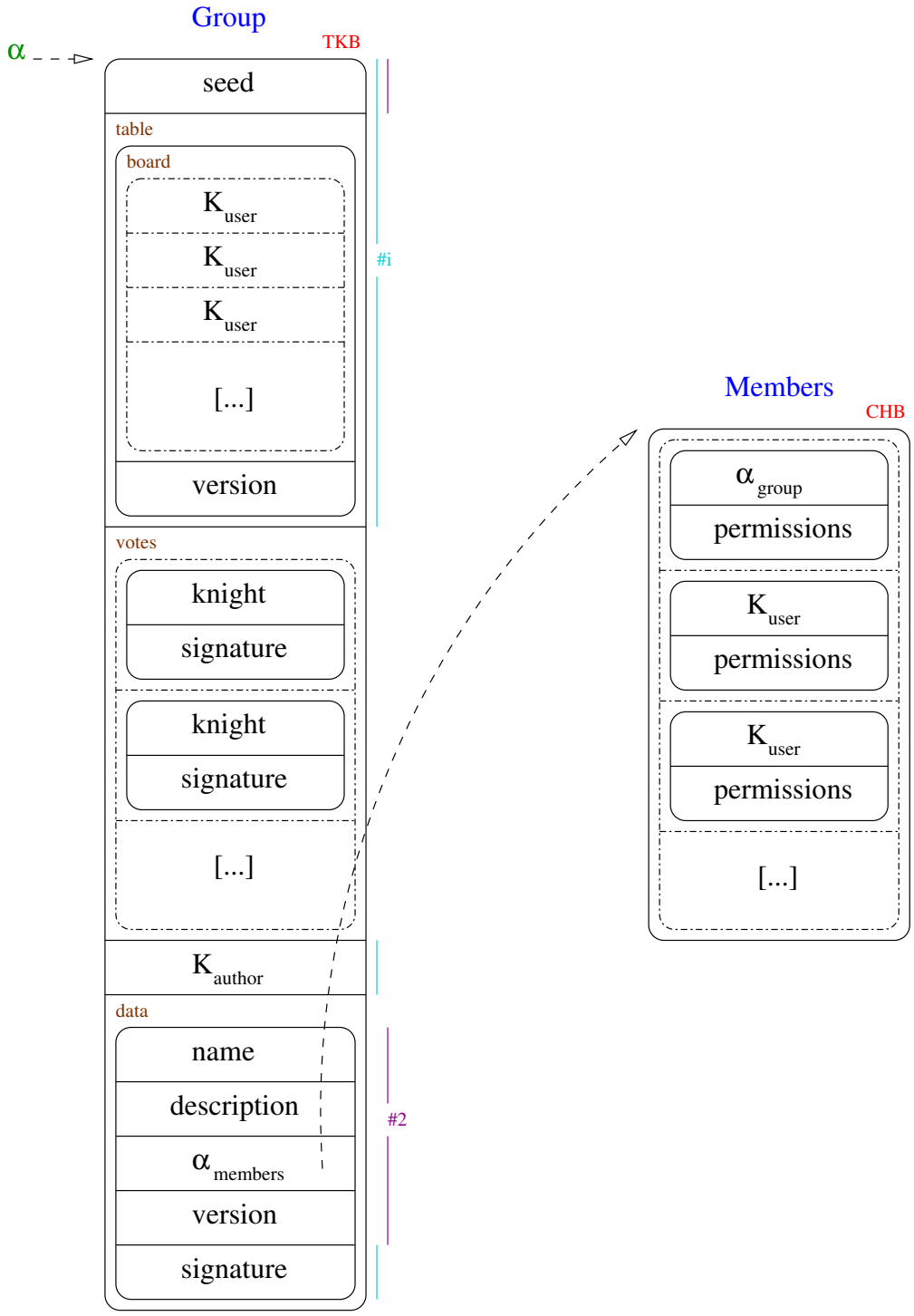


Figure 4.11: The representation of an *TKB*-based *Group* block

4.2.4.2 Ownership

The other mechanism included in the administration scheme focuses on providing users the possibility to transfer their ownership to another user, should this user agree to take over this responsibility.

∇_1: environment, ∇_2: modification, ∇_3: consultation, ∇_4: space, ∇_5: flexibility, ∇_6: efficiency, ∇_7: accountability

Interestingly, the previously described *TKB* physical block seems to provide the ideal construct for this functionality. Indeed, every *Object* could be built upon a *TKB* such that the table of knights would be composed of a single user, the block's owner. Unlike *OKB*-based *Object*s, such blocks would not be statically bound to their owner since the table of knights can evolve over time. This characteristic makes the transfer of ownership achievable through the sole modification of the table of knights' composition.

Although this model would provide the required functionality, every file system object whose ownership may eventually be transferred would have to be built on such a physical block. Unfortunately, such blocks may perform extremely poorly depending on the owner's connectivity. Indeed, since every block modification must be approved by the table of knights, which in this case is composed of a single user, users wanting to update the block may fail to do so because of the owner's unavailability to validate the operation.

A solution could be to introduce another physical block benefiting from both *OKB*s' and *TKB*s' advantages. Such a physical block would embed a table of knights for moderating operations regarding the modifications of the block's metadata including the composition of the table of knights. However, the data modifications would not require knights' approval. Therefore, such a physical block would perform exactly as *OKB*s though relying on a table of knights. Besides, such a model, applied to *Object* logical blocks, could benefit from the lords' connectivity, as discussed in *Sections 4.1.3 & 4.1.7*.

Unfortunately, while this model implies the introduction of another physical block, hence increasing the system's complexity, it also fails to fulfill the objective: the future owner must accept to take over the ownership. Since the future owner must be included in the process, a user-to-user protocol is absolutely necessary.

The remainder details an extremely simple user-to-user protocol which assumes that every file system object is stored in an *OKB* physical block, except for community objects such as the root directory, the users inventory and so forth which would rely on *TKB*s.

Since *OKB*s are bound to their respective owners, transferring the ownership implies creating a new *OKB*. However, although the content of the *Object* block must be cloned, the *Contents* block, the *Access* block as well as the data blocks can be re-used. As such, the *Object* block's content is copied into another *Object* while the new owner's public key is inserted and the metadata section's signature is re-issued, by the new owner. This straightforward process is therefore very efficient: *(i)* a single key pair must be generated *(ii)* a single block must be cloned and *(iii)* two cryptographic signatures must be recomputed.

Noteworthy is that this method implies several additional actions to be performed.

∇_1: environment, ∇_2: modification, ∇_3: consultation, ∇_4: space, ∇_5: flexibility, ∇_6: efficiency, ∇_7: accountability

Firstly, the old *Object* block must be deleted though the blocks it references such as the *Access*, *Contents* and data blocks must not. Note however that a garbage collection mechanism would not require the original owner to do anything as the system would detect the block as no longer used and it would therefore be automatically swept out. Secondly, every directory referencing the old *Object* block must be updated with the address of the new *Object*.

The protocol, detailed below, enables a user to transfer the ownership of the block identified by its address α to another user μ.

1. First, the *Object*'s current owner sends the message $\langle \texttt{transfer}, \alpha, \kappa^{\widetilde{K_\mu}} \rangle$ to the potentially future owner μ.

   Note that κ, the data encryption key, is provided so that the future owner can access the data. However, the key is encrypted with the μ user's public key in order to ensure security.

2. The user receiving such a message decides whether or not to accept the ownership. If so, the block is cloned through the creation of another *Object* block and updated accordingly.

   Then, the message $\langle \texttt{accept}, \alpha, \epsilon \rangle$ is sent back to the original owner. The message includes ϵ, the address of the cloned *Object* block.

3. Finally, the original owner, having received the `accept` message, updates every directory referencing the *Object* block with the new address ϵ.

This protocol is particularly interesting because it can be applied to every physical block. Thus, a user could decide to transfer the ownership not to another user but instead to a community so that a *TKB* would be created to replace the *OKB*. Similarly, a community could decide that the object no longer needs several users to monitor the object but instead that a single user should be made owner, hence transforming a *TKB* into an *OKB*.

4.2.5 Algorithms

This section provides the reader a complete set of algorithms for manipulating *TKB*-based *Object* and *Group* logical blocks.

- $\texttt{Elect}(\alpha, \theta)$
- $\texttt{Govern}(\alpha, \psi)$
- $\texttt{Read}(\alpha, \lambda) \longrightarrow \delta$

- `Write`$(\alpha, \lambda, \delta)$
- `Manage`(α, θ)
- `Edit`(α, ψ)
- `Transfer`(α, K_{user})

Note that, as described earlier, every *TKB*-related `Put()` request must provide both the block to store and a set of votes issued by the current knights. This action is symbolised next by the | operator which concatenates the block with the additional votes so that every storage node can extract the necessary information and perform the server-specific verification process.

Although every algorithm described below makes use of timeouts, no information is given regarding the mechanism to handle such timeouts. The reader should consider that clients are supposed to retry the operation later and eventually return an error after a certain number of successive failures.

Elect

The `Elect`(α, θ) routine takes the address α of an *Object* object along with a set of knights θ and requests that the table's composition be changed. The elect permission authorising users to perform this operation can be granted to anyone, directly in the *Access* block associated with the *Object* or indirectly through a chain of group memberships, assuming that every group of the chain is granted the elect permission along with the user.

Algorithm 28 illustrates the client's process which consists in acquiring votes from the current knights in order to prove the storage nodes that the operation has been approved but also from the future knights in order to build a valid block.

Require: (K_{user}, k_{user}), the user's personal key pair

1. $\beta \leftarrow \texttt{Gather}(\alpha)$
2. **for all** $\phi \in \beta.table.board$ **do**
3. **request** a vote from knight ϕ by sending the message $\langle \texttt{elect}, \alpha, K_{user}, \theta \rangle$
4. **end for**
5. $\varepsilon \leftarrow$ **wait** until $\left\lceil \frac{|\beta.table.board|}{2} \right\rceil$ votes have been received, or timeout
6. **for all** $\phi \in \theta$ **do**
7. **request** a vote from knight ϕ by sending the message $\langle \texttt{elect}, \alpha, K_{user}, \theta \rangle$
8. **end for**
9. $\upsilon \leftarrow$ **wait** until $\left\lceil \frac{|\theta|}{2} \right\rceil$ votes have been received, or timeout
10. $\beta.votes \leftarrow \upsilon$
11. $\beta.table.board \leftarrow \theta$
12. $\beta.table.version \leftarrow \beta.table.version + 1$
13. $\texttt{Seal}_{\frac{Object}{TKB}}(\alpha, \beta)$
14. $\texttt{Put}(\alpha, \beta|\varepsilon)$ – *the | operator designates concatenation*

Algorithm 28: $\texttt{Elect}(\alpha, \theta)$

Govern

The $\texttt{Govern}(\alpha, \psi)$ routine enables any user having been granted the govern permission to update the *Object*'s metadata, including the access control list in the *Access* block.

The function takes the address of the *Object* block along with a set of modifications to apply on the metadata.

Require: (K_{user}, k_{user}), the user's personal key pair

1. $\beta \leftarrow \texttt{Gather}(\alpha)$
2. **for all** $\phi \in \beta.table.board$ **do**
3. **request** a vote from knight ϕ by sending the message $\langle \texttt{govern}, \alpha, K_{user}, \psi \rangle$
4. **end for**
5. $\upsilon \leftarrow$ **wait** until $\left\lceil \frac{|\beta.table.board|}{2} \right\rceil$ votes have been received, or timeout
6. $\beta.votes \leftarrow \upsilon$
7. $\beta.meta.version \leftarrow \beta.meta.version + 1$
8. **apply** the set of modifications ψ
9. **for all** (ϵ, ζ) such that ζ is a new immutable block **do**
10. $\texttt{Seal}_{\frac{}{CHB}}(\epsilon, \zeta)$
11. $\texttt{Put}(\epsilon, \zeta)$
12. **end for**
13. $\texttt{Seal}_{\frac{Object[meta]}{TKB}}(\alpha, \beta)$
14. $\texttt{Put}(\alpha, \beta|\upsilon)$ – *the | operator designates concatenation*

Algorithm 29: $\texttt{Govern}(\alpha, \psi)$

Read

The $\texttt{Read}(\alpha, \lambda, \delta)$ method is very similar to the *OKB*-based *Object*-specific *Algorithm 18* except that there is no special case for the owner.

Let us recall that vassals are no longer required to acquire the validation of their modification from a lord as it was with *OKB*s. Instead, in the context of *TKB*s, the knights perform this verification for every modification, no matter the role of the author: knight, lord or vassal. However, vassals must still contact a lord when it comes to reading the data in order to retrieve the key required to decrypt the *Object*'s *Contents* block.

Require: (K_{user}, k_{user}), the user's personal key pair

1. $\beta \leftarrow \texttt{Gather}(\alpha)$
2. $\chi \leftarrow \texttt{Get}(\beta.meta.\alpha_{access})$
3. **if** $\exists \iota \ : \ K_{user} = \chi[\iota].K_{user} \quad \forall \iota \in \left\{1, ..., |\chi|\right\}$ **then**
4. **if** $read \notin \chi[\iota].permissions$ **then**
5. **error** *"the lord does not have the read permission"*
6. **end if**
7. $\kappa \leftarrow \chi[\iota].token^{\widetilde{k_{user}}}$
8. **else**
9. $\Xi \leftarrow \left\{\chi[\iota].K_{user} \ : \ read \in \chi[\iota].permissions \quad \forall \iota \in \left\{1, ..., |\chi|\right\}\right\}$
10. $\nu \leftarrow \texttt{Locate}(\Xi)$
11. $\kappa \leftarrow$ **request** user ν for the key by sending the message $\langle \texttt{read}, \alpha, K_{user} \rangle$
12. **end if**
13. **if** $h(\kappa) \neq \beta.data.fingerprint$ **then**
14. **error** *"the key does not match the fingerprint"*
15. **end if**
16. $\sigma \leftarrow \texttt{Get}(\beta.data.\alpha_{contents})$
17. $\xi \leftarrow$ **decrypt** σ with the key κ
18. $\delta \leftarrow$ **read** data from ξ at location λ
19. **return** δ

Algorithm 30: $\texttt{Read}(\alpha, \lambda) \longrightarrow \delta$

Write

The $\texttt{Write}(\alpha, \lambda, \delta)$ routine provides authorised users the possibility to request the modification of the *Object*'s data. Unlike *OKB*-based *Object*s, both lords and vassals must acquire the approval from the table of knights.

As for elect, govern and read, the write permission can be granted to anyone, including group members.

Require: (K_{user}, k_{user}), the user's personal key pair

1. $\beta \leftarrow \texttt{Gather}(\alpha)$
2. $\chi \leftarrow \texttt{Get}(\beta.meta.\alpha_{access})$
3. **for all** $\phi \in \beta.table.board$ **do**
4. **request** a vote from knight ϕ by sending the message $\langle \texttt{write}, \alpha, K_{user}, \delta \rangle$
5. **end for**
6. $\upsilon \leftarrow$ **wait** until $\left\lceil \frac{|\beta.table.board|}{2} \right\rceil$ votes have been received, or timeout
7. $\beta.votes \leftarrow \upsilon$
8. $\beta.data.version \leftarrow \beta.data.version + 1$
9. $\kappa \leftarrow$ **generate** cryptographic symmetric key
10. $\xi \leftarrow$ **write** data δ at location λ
11. $\sigma \leftarrow$ **encrypt** ξ with the key κ
12. $\beta.data.\alpha_{contents} \leftarrow \texttt{Setup}_{\overline{CHB}}(\sigma)$
13. $\texttt{Seal}_{\overline{CHB}}(\beta.data.\alpha_{contents}, \sigma)$
14. $\texttt{Put}(\beta.data.\alpha_{contents}, \sigma)$
15. **for all** $\iota \in \left\{1, ..., |\chi|\right\}$: $read \in \chi[\iota].permissions$ **do**
16. $\chi[\iota].token \leftarrow \kappa^{\overbrace{\chi[\iota].K_{user}}}$
17. **end for**
18. $\beta.data.fingerprint \leftarrow h(\kappa)$
19. **for all** (ϵ, ζ) such that ζ is a new immutable block **do**
20. $\texttt{Seal}_{\overline{CHB}}(\epsilon, \zeta)$
21. $\texttt{Put}(\epsilon, \zeta)$
22. **end for**
23. $\texttt{Seal}_{\frac{Object[data]}{TKB}}(\alpha, \beta)$
24. $\texttt{Put}(\alpha, \beta|\upsilon)$ – *the | operator designates concatenation*

Algorithm 31: $\texttt{Write}(\alpha, \lambda, \delta)$

Manage

The $\texttt{Manage}(\alpha, \theta)$ routine is equivalent to the $\texttt{Elect}(\alpha, \theta)$ method in every aspect except that it operates on a *Group* block. Note however that this permission cannot be granted to a user or group in an *Access* block since the operation is not related to an *Object*.

As previously mentioned, distinguishing the manage from the elect permission enables a group member to be given the right to modify the table of knights of both the user's *Group* and an *Object* the group has been granted the permission to.

Require: (K_{user}, k_{user}), the user's personal key pair

1. $\beta \leftarrow \texttt{Gather}(\alpha)$
2. **for all** $\phi \in \beta.table.board$ **do**
3. **request** a vote from knight ϕ by sending the message $\langle \texttt{manage}, \alpha, K_{user}, \theta \rangle$
4. **end for**
5. $\varepsilon \leftarrow$ **wait** until $\left\lceil \frac{|\beta.table.board|}{2} \right\rceil$ votes have been received, or timeout
6. **for all** $\phi \in \theta$ **do**
7. **request** a vote from knight ϕ by sending the message $\langle \texttt{elect}, \alpha, K_{user}, \theta \rangle$
8. **end for**
9. $\upsilon \leftarrow$ **wait** until $\left\lceil \frac{|\theta|}{2} \right\rceil$ votes have been received, or timeout
10. $\beta.votes \leftarrow \upsilon$
11. $\beta.table.board \leftarrow \theta$
12. $\beta.table.version \leftarrow \beta.table.version + 1$
13. $\texttt{Seal}_{\frac{Group}{TKB}}(\alpha, \beta)$
14. $\texttt{Put}(\alpha, \beta|\varepsilon)$ – *the | operator designates concatenation*

Algorithm 32: $\texttt{Manage}(\alpha, \theta)$

Edit

The $\texttt{Edit}(\alpha, \psi)$ routine takes the address of a *Group* block along with a set of metadata modifications. As for manage, the edit permission can only be granted to group members, granting them the right to request some modifications, including the set of members and their permissions.

Require: (K_{user}, k_{user}), the user's personal key pair

1. $\beta \leftarrow \texttt{Gather}(\alpha)$
2. **for all** $\phi \in \beta.table.board$ **do**
3. **request** a vote from knight ϕ by sending the message $\langle \texttt{edit}, \alpha, K_{user}, \psi \rangle$
4. **end for**
5. $\upsilon \leftarrow$ **wait** until $\left\lceil \frac{|\beta.table.board|}{2} \right\rceil$ votes have been received, or timeout
6. $\beta.votes \leftarrow \upsilon$
7. $\beta.data.version \leftarrow \beta.data.version + 1$
8. **apply** the set of modifications ψ
9. **for all** (ϵ, ζ) such that ζ is a new immutable block **do**
10. $\texttt{Seal}_{\overline{CHB}}(\epsilon, \zeta)$
11. $\texttt{Put}(\epsilon, \zeta)$
12. **end for**
13. $\texttt{Seal}_{\overline{TKB}}^{Group}(\alpha, \beta)$
14. $\texttt{Put}(\alpha, \beta|\upsilon)$ – *the | operator designates concatenation*

Algorithm 33: $\texttt{Edit}(\alpha, \psi)$

Transfer

The $\texttt{Transfer}(\alpha, \mu)$ routine enables a user owning an *OKB*-based *Object* to transfer her ownership to another user.

The routine takes the address α of an *Object* block along with the identity μ of the user to whom the ownership is to be offered.

Require: (K_{user}, k_{user}), the user's personal key pair

1. $\beta \leftarrow \texttt{Gather}(\alpha)$
2. **if** $K_{user} \neq \beta.owner.K_{owner}$ **then**
3. **error** *"the owner is the only user authorised to transfer the object's ownership"*
4. **end if**
5. $\kappa \leftarrow \beta.meta.owner.token^{\widetilde{k_{user}}}$
6. **request** user μ to take over by sending the message $\langle \texttt{transfer}, \alpha, \kappa \rangle$
7. **wait** for message $\langle \texttt{accept}, \alpha, \epsilon \rangle$, or timeout
8. **update** the referencing directories with the address ϵ

Algorithm 34: $\texttt{Transfer}(\alpha, \mu)$

4.2.6 Analysis

The previous sections detailed the design of an administration scheme composed of a community and ownership mechanism. Although the community mechanism enables objects to be managed with great flexibility, it also introduces numerous inconveniences which are discussed below.

The *TKB*-specialized quorums have been optimised in order to prioritise reads over writes. Firstly, readings have been assumed to be more common than writings. Although this assumption seems fair, some specific file systems may wish to optimise the file system the other way around. Fortunately, quorum algorithms are highly adaptable and could therefore be optimised for writings: write quorums would require the acquisition of $2\gamma + 1$ nodes while read quorums would be made more expensive with $3\gamma + 1$ nodes. Secondly, *TKB* physical blocks require every modification to be approved by acquiring knights' vote, process which can drastically delay the operation from taking effect. On the other hand, read operations can be performed without such approvals though vassals may have to contact another user to retrieve the key. Therefore, while write operations are prone to large delays, readings can be performed in real-time, so to speak: in the worst case scenario a single user's node must be contacted.

The delay potentially implied by write quorums comes from the fact that the $3\gamma + 1$ storage nodes may not be operational at the time of the acquisition, hence making the writing impossible to complete. However, since the underlying network protocols have been designed with self-adaptability in mind, other nodes will be chosen to take over the non-responsive ones such that, within a few seconds, a write quorum can be expected to be acquired. Besides, let us recall that writings require users to request votes from the table of knights, a process which could take from a few minutes to several days. Unfortunately, this delay cannot be determined because independent from the system. Indeed, should a majority of the knights of a single table be absent for several days, every request made during this period would be delayed until the knights come back and process the requests manually. As a result, the delay inherent in write quorums can be ignored in regard to the expensive votes acquisition process. Finally, note that most implementations purposely delay the commit of write operations. This technique is often used to buffer successive writes but also to avoid network communications should a file be created and immediately destroyed, operations which are very common in the process of compiling source code files for instance.

For all the reasons exposed above, the *TKB*-specialized quorum algorithm has been optimised for reading rather than writing. Note that the common quorum algorithm used for other mutable blocks such as *OKB*s and *PKB*s has not been been optimised for one or the other type of operation because both can be performed in real-time *i.e.*

without the approval of some authority. However, the file system could very much be configured for optimising reading for instance, in which case read quorums would require the acquisition of $\gamma + 1$ instances while write quorums would be composed of $3\gamma + 1$ storage nodes.

The *TKB* physical block has been designed for the specific purpose of administration and should not be used otherwise. For instance, it would be inappropriate to build an entire file system out of *TKB*s. Indeed, as most of the file system objects are administered in a discretionary manner [Vog99], *Object* blocks would embed a table of knights composed of a single user. Since approval is required to update such blocks, the connectivity of the knights would thus be absolutely crucial to the system. Unfortunately, *Section 4.1.7* showed that the connectivity of a single user would not suffice to ensure users the possibility to interact normally with objects.

Although *TKB*s suffer the exact same connectivity issue as *OKB*s, both blocks exhibit different characteristics. Firstly, *OKB*s require vassals to request a lord in order to operate on the object, both for reading and writing. *TKB*s, on the other hand, require every user wanting to update the block to contact knights while vassals have to request a lord for the encryption key when it comes to read operations. Secondly, while a single lord was enough to perform an *OKB*-related operation, at least half of the knights must be contacted in order to achieve consensus. Thirdly, the users acting as an *OKB*'s lords were designated by the block's owner without their accord. However, every user acting as a *TKB*'s knight chooses this position knowing exactly what it implies. Therefore, knights can be expected to be extremely well-connected users, probably responding to every administrative request within a few hours. Besides, the table of knights has been designed to be self-moderated so that a knight failing to perform her duty could be evicted by the community itself.

The client's connectivity also plays an important role in the votes acquisition process. Assuming that some knights were disconnected at the time the client made its request, the client should periodically try to resend the request so that every knight eventually receives it. Besides, the client must be connected to the network in order to collect the votes from the multiple knights.

To summarise, the *TKB* physical block is costly to manipulate because *(i)* the *TKB*-specific quorums are expensive *(ii)* writes may suffer from delay, from a few minutes to several days *(iii)* the block contains more information than *OKB*s, especially because of the embedded table of knights along with the attached votes and *(iv)* *TKB*s are more expensive to validate than *OKB*s because every vote includes a cryptographic signature.

Unfortunately, *TKB*'s disposition for inducing delays in write operations drastically increases concurrency issues. As for *OKB*s, multiple users may concurrently update a block from version ν_i to ν_{i+1} implying that the first one to commit the modifications

would automatically render the other ones invalid[3]. Alas, while the delay between a block retrieval and the commit of its modifications may approximate several seconds for *OKB*s, it could approach several days for *TKB*s. The probability of such update collisions are therefore extremely high, especially since *TKB*s will probably be used for popular content such as the users inventory, object which should be modified on a regular basis. Noteworthy is that agreement protocols such as *BFT* [CL99], *Paxos* [Lam98] *etc.* do not suffer from such a limitation because such algorithms ensure serializability.

To conclude, although the community mechanism has been designed to prevent chaos, especially regarding critical objects such as the root directory, the user inventory and so on, the design's extreme flexibility provides any set of users the possibility to manage a block in a cooperative manner. Furthermore, it is interesting to notice that *Section 4.1.2* discussed the various access control paradigms, from *MAC* and *DAC* to *RBAC*, discarding both *MAC* and *RBAC* models because they require system-wide definitions. Ironically, the community mechanism designed throughout this section shows how such system-wide definitions could actually be achieved through the use of a dedicated community deciding whether or not to create new roles or to include a user in a higher clearance level for instance. Note however that although such paradigms could be designed through this framework, they would not resolve the fundamental flaws that have been discussed throughout this chapter.

⋆ ⋆
⋆

This chapter focused on designing the key components required in order to build a peer-to-peer file system. The first section introduced the notion of user but also presented the design of the *Object* block abstraction upon which every file system object can be built. Also, a flexible access control scheme has been integrated, providing users the means to protect or share information with other users and/or groups. The second section discussed several organisation models leading to the conclusion that peer-to-peer file systems, as common centralised file systems, need to provide an administration scheme. The proposed scheme is composed of both a community and ownership mechanism enabling communities to manage blocks in a cooperative manner.

[3]Let us recall that the version numbers must increase in a strictly monotonic way.

The contributions of this design are threefold. Firstly, the access control scheme enables users to express access restrictions in a flexible way that is unprecedented. Secondly, this work seems to be the first, as far as the author is aware, to address the issues related to providing an administration mechanism in such a decentralised environment. Thirdly, an extremely simple yet efficient user-to-user protocol enables users to transfer their ownership to other users.

However, the design does not come without trade-offs. First of all, the connectivity requirement has been loosened against the fundamental properties defined in *Chapter 3*, in order to achieve the required flexibility. Note however that this constraint is minor, especially compared to other projects [KRS+03], as the study carried out throughout *Sections 4.1.7 & 4.2.6* showed that a set of five lords with a connectivity ratio of 0.6 suffices to enable vassals to interact with the object. Finally, the major drawback lies in the community mechanism's *TKB* physical block which, due to its tendency to large delays, drastically increases the probability of conflicting concurrent updates.

Chapter 5

Implementation

This chapter details the implementation of the *Infinit* peer-to-peer file system which follows the philosophy of *WheelFS* [Sa07] by focusing on providing the fundamental functionalities such as access control and administration through a file system abstraction.

The *Infinit* prototype is composed of $40,000$ lines of source code written in *C++*. The implementation relies on several libraries, mainly the *STL (Standard Template Library)* for common system features and the *OpenSSL (Open Secure Socket Layer)* for cryptographic capabilities. Besides, since the file system has been written in *C++*, the *Infinit* processes reside in userspace. Therefore, the implementation relies on *FUSE (File System in User Space)* [FUS] for forwarding the system calls from the operating system kernel to a specific *Infinit* process.

Note that although *Infinit* provides a completely useable file system abstraction, many components have been implemented in their simplest possible form in order to meet the time constraints.

5.1 Representation

The file system prototype relies, as defined in *Chapter 3*, on a distributed storage layer which ensures some fundamental network properties such as scalability, durability, availability and so forth. It is important to note that, as for *CFS* [DKK+01], *Plutus* [KRS+03] or even *Pastis* [mBPS05], the data being from a file, directory or else are split into blocks in order to better balance the storage load between the nodes.

Infinit follows the *UFS (UNIX File System)* organisation in which every file system object such as files, directories *etc.* are represented by an *inode*, though *Infinit* uses a slightly different terminology, detailed next.

File objects are represented through the logical block *Object* which contains, as described throughout *Chapter 4*, the metadata. Also, the *Access* block contains the access control rules enabling the object owner to restrict and/or share access to other users. However, as described in *Section 4.1*, the access control scheme provides far more expressivity than the *UNIX* permissions. Finally, the *Contents* block contains references to the set of encrypted *Data* blocks. The relations between these logical blocks, forming a so-called file, are illustrated by *Figure 5.1*.

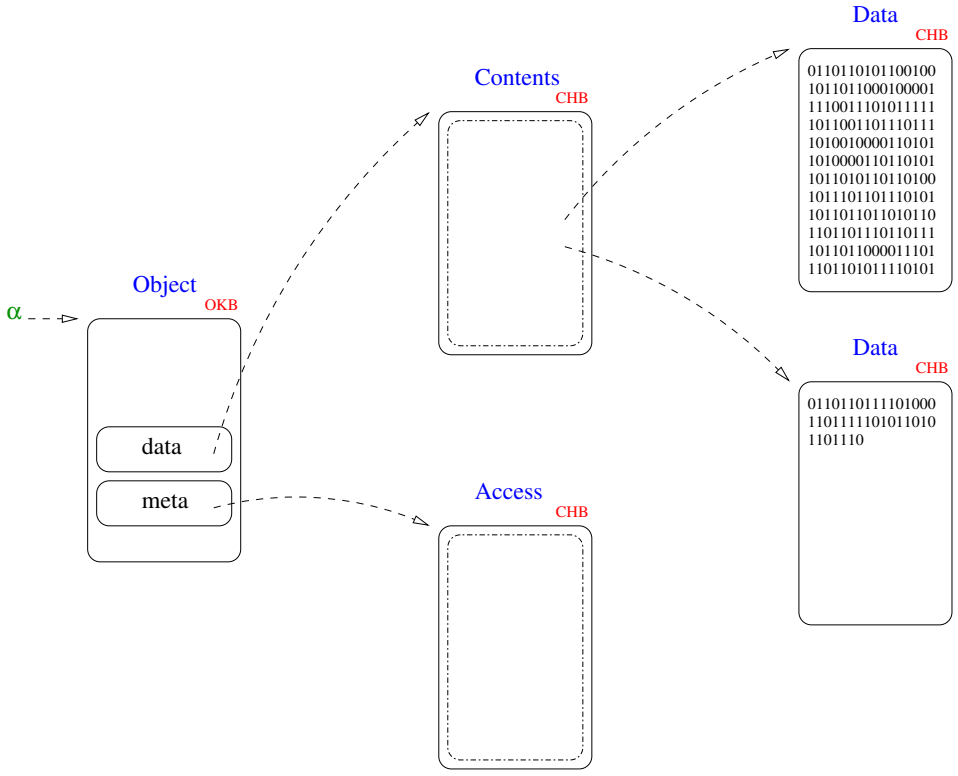


Figure 5.1: A file representation

Directory objects are similar to files in every aspect except regarding the data semantics. The directory entries are stored in *Catalog* logical blocks, as shown on *Figure 5.2*. Note that every directory entry is composed of the name of the referenced object along with the address of the pointed to *Object* block, being a file, sub-directory *etc.*

In addition, *Infinit* provides users with objects commonly referred to as links. However, one should carefully note that these objects relate to *UNIX* symbolic rather than hard links. Indeed, since every *Infinit* file system object is administered in a discretionary manner, hard links would be impossible to implement, since a counter should be updated following every hard link creation and destruction, not mentioning inconsistencies inherent to malevolent behaviours.

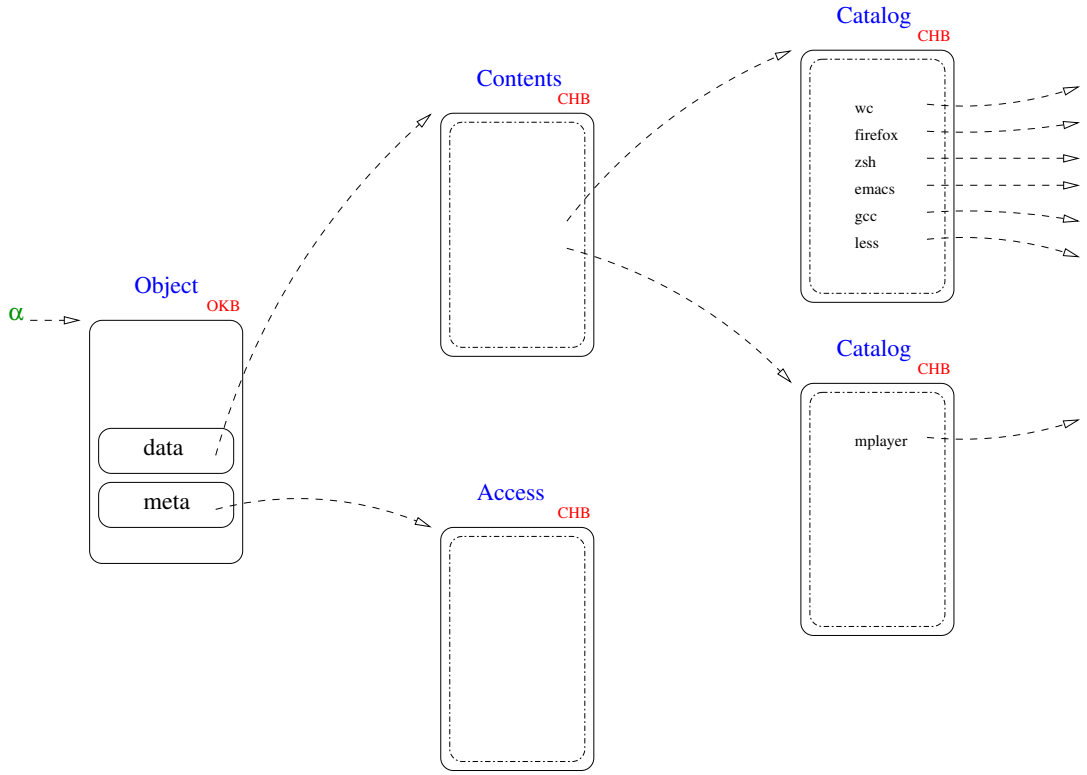


Figure 5.2: A directory representation

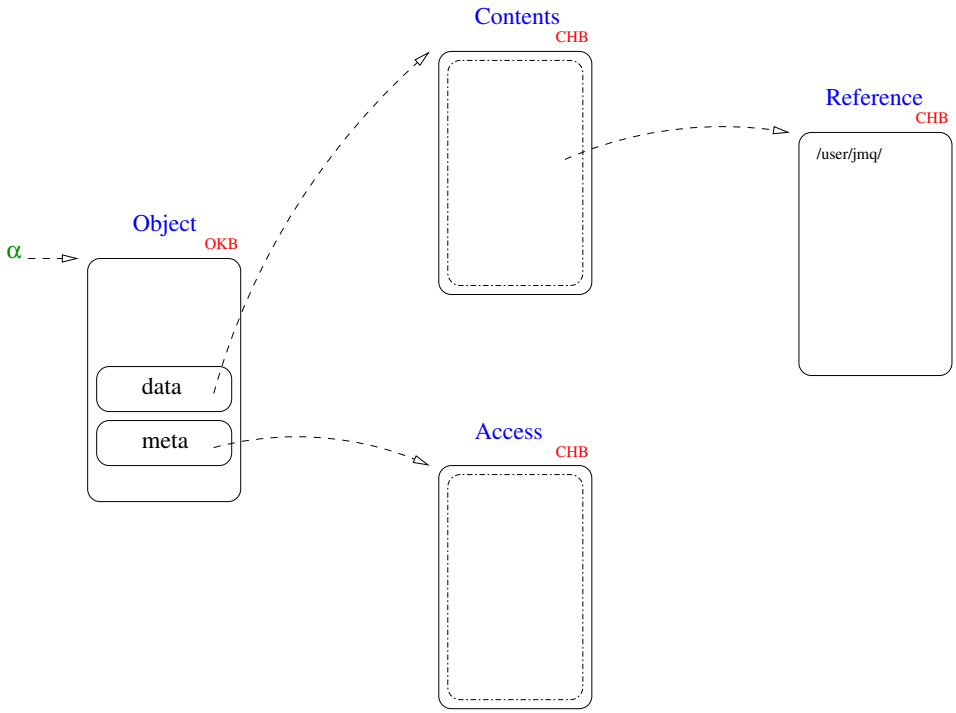


Figure 5.3: A link representation

The *Reference* block contains the path indicating the namespace location of the

target object. Noteworthy is that, should the path be extremely long, it would actually be split over several *Reference* blocks chained and referenced by the object's *Contents* block.

Figure 5.3 illustrates these relations although, in this example, the path is short enough to fit in a single *Reference* block.

The prototype makes use of a system-wide organisation model, meaning that a single hierarchical namespace is exposed to the users, independently of their preferences. As discussed in *Section 4.2.2*, the system-wide organisation model may suffer from inconsistencies, though they are less likely to occur than in the user-wide model. Besides, this model is far easier to implement than a user-wide scheme, which would require the development of an extended set of applications for manipulating views.

```
1   /
2     users
3     music/
4       Tool/
5       Camel/
6       Magma/
7       ...
8     README
```

Listing 5.1: An example of hierarchical namespace

Figure 5.4 depicts the relations between the various logical blocks composing the hierarchical namespace given by *Listing 5.1*. This example illustrates the use of the community mechanism described in *Section 4.2*. Indeed, both the root directory and the `/users` file rely on the *TKB* construct. The `/users` special file is assumed, in this context, to act as a user inventory in which every entry is composed of a name and the address of the associated *User* logical block. Note that for the sake of clarity, the *Contents* and *Access* blocks have been omitted from *Figure 5.4*.

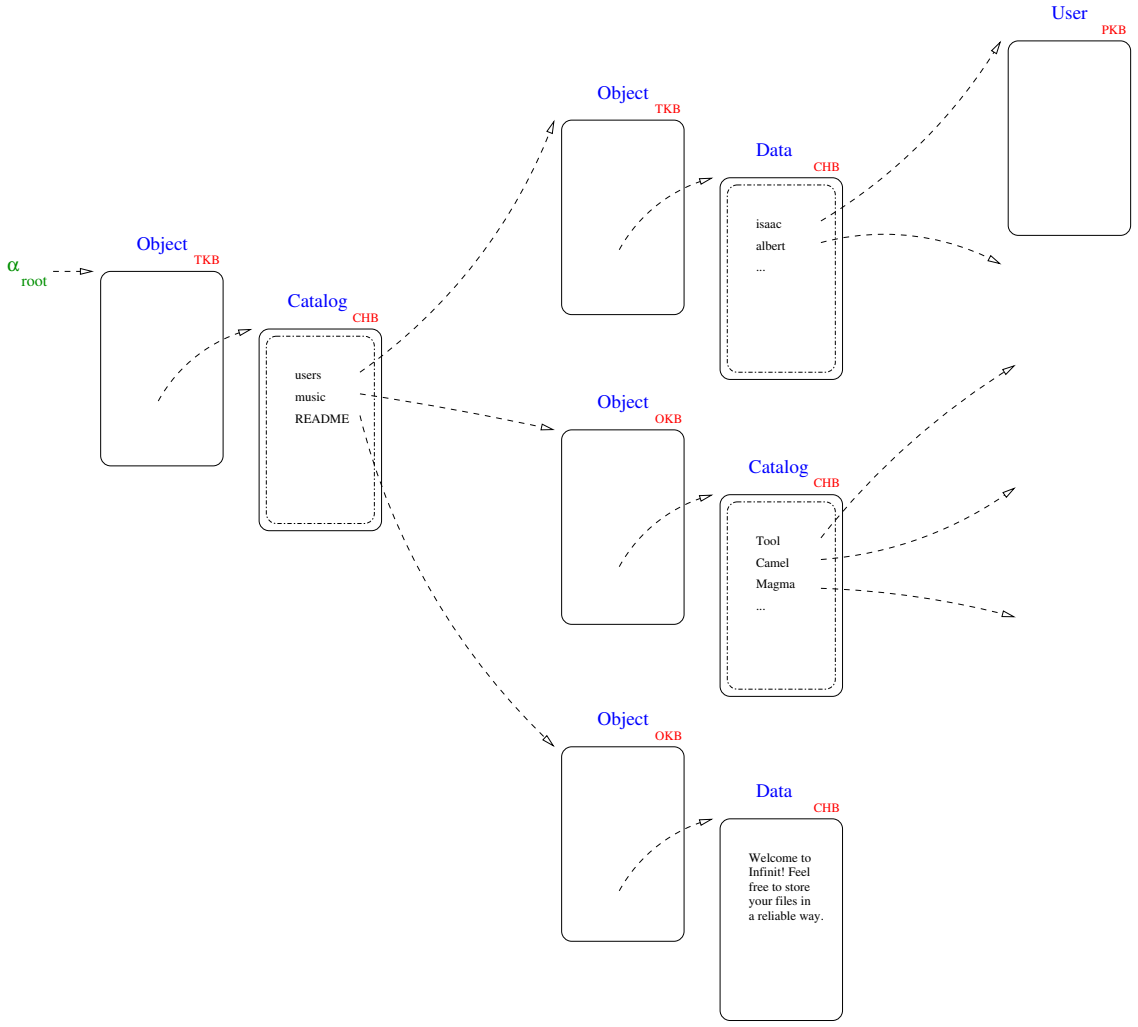


Figure 5.4: The *Infinit* system-wide hierarchical representation

5.2 Architecture

The architecture of *Infinit* has been broken down into small, coherent, and flexible components known as *Agent*, *Etoile*, *PIG* and *Hole*. In addition, every component relies on one or both of the *Elle* and *Lune* libraries. All these components are described below.

Figure 5.5 details the components composing a node connecting to the *Infinit* peer-to-peer network. Whenever a user, through an application, performs a file system-related operation on an *Infinit* partition, the standard *C Library* issues a system call to the operating system kernel. The kernel, noticing that *Infinit* is a userspace file system, forwards the call to the *PIG (POSIX/Infinit Gateway)* component. *PIG* aims at transcoding *POSIX* file system calls by sending the corresponding request to *Etoile*. Note that whenever a cryptographic operation must be performed on behalf of the requesting user, *Etoile* sends a message to the *Agent* component whose

purpose is to handle operations such as signing and decrypting data with the user's personal key pair. Finally, should *Etoile* require to update or retrieve information from the distributed hash table, a request is sent to *Hole* which acts as the gate to the peer-to-peer network.

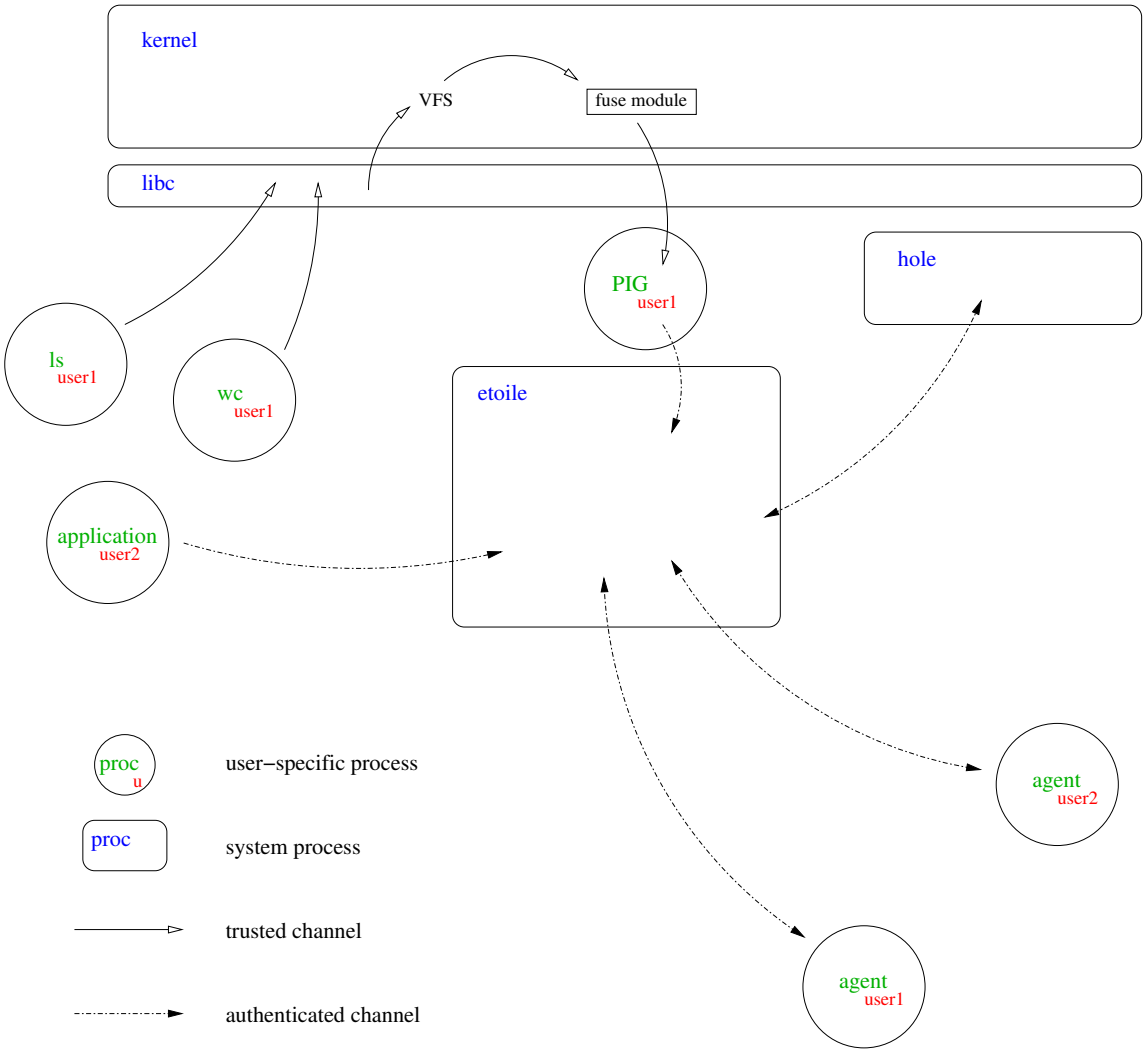


Figure 5.5: The architecture of an *Infinit* node

Noteworthy is that *Infinit* can be configured depending on the node's purpose. For instance, one may want to set up *Infinit* on an enterprise server in order to contribute the server's storage capacity to the company's private *Infinit* network. However, since nobody is using the server as a personal computer, the components *PIG*, *Agent* and *Etoile* can be omitted. Similarly, one might be willing to configure *Infinit*, on a mobile phone, so that whenever *Etoile* requires *Agent* to perform a cryptographic operation, the request is sent to the user's home desktop computer rather than to an *Agent* running on the mobile phone. The decomposition of *Infinit* into several units enables users to configure the software depending both on their preferences but

also on the device's specifics, hence complying with the mobility property defined in *Chapter 3*.

The remainder of this section discusses the internals of every component, including the libraries.

5.2.1 Elle

Elle is a general-purpose library used by all the other components of the *Infinit* file system. *Elle* provides several packages, each of which offers a specific set of functionalities. The most important of those packages are described next.

core

The *core* package contains a set of basic types on which everything else is built. These types were introduced in order to ease the process of data serialisation. Those includes `Boolean`, `Byte`, `Integer8`, `Integer64`, `Natural32`, `Real` among others.

system

The *system* package contains system-specific definitions such as the microprocessor's endianness, the path to major locations including the user's local home directory, the local root directory *etc.*

standalone

The *standalone* package is very specific as it provides fundamental functionalities which may require features provided by other *Elle* packages, leading to inter-dependencies. The classes provided by this package were thus implemented by sometimes relying on code redundancy.

The package offers functionalities such as `Region` for manipulating static and dynamic buffers, `Report` for handling error messages or even `Maid` for automatic memory deallocation *etc.*

radix

The *radix* package contains fundamental base classes. Firstly, the `Meta` class represents the ultimate base class which is directly or indirectly inherited by any other class. Secondly, every class must inherit one of the following classes: `Object` or `Entity`.

The `Object` class must be inherited by any class which describes an object that can be serialised, compared, cloned, recycled and so on. On the other hand, any other class must inherit the `Entity` class.

archive

The *archive* package contains the `Archive` class which provides methods for serialising and extracting the types defined in the *core* package. Since most objects are built upon such types, an additional mechanism is required to render these types serialisable.

Every class inheriting the `Object` class implicitely derives `Archivable`. Such classes must implement the associated `Serialize()` and `Extract()` methods which specify the internal attributes to be included in the serialisation process.

```
1    Status              PublicKey::Serialize(Archive&   archive) const
2    {
3      // serialize the internal numbers.
4      if (archive.Serialize(*this->key->pkey.rsa->n,
5                            *this->key->pkey.rsa->e) == StatusError)
6        escape('unable to serialize the internal numbers');
7
8      return (StatusOk);
9    }
```

Listing 5.2: The `PublicKey::Serialize()` method

```
1    Status              KeyPair::Extract(Archive&       archive)
2    {
3      // extract the internal keys.
4      if (archive.Extract(this->K, this->k) == StatusError)
5        escape('unable to extract the internal keys');
6
7      return (StatusOk);
8    }
```

Listing 5.3: The `KeyPair::Extract()` method

Listing 5.2 illustrates the implementation of the `Serialize()` method for the specific purpose of the `PublicKey` class. Similarly, *Listing 5.3* depicts the opposite process through the `Extract()` implementation for the `KeyPair` class. The reader can notice that the terminology—K and k for the public and private keys, respectively—is equivalent to the notation introduced in *Chapter 4*.

io

The *io* package provides basic input/output classes and interfaces such as `Fileable` and `Dumpable`. The most notable `Uniquable` interface enables any deriving class to represent objects in the form of a unique literal string. *Listing 5.4* illustrates the use of the `Uniquable` interface for the address of a block. Also, the original object can be reconstructed from a `Unique` representation making such strings extremely convenient as human-readable tokens.

```
1 RUxMAQkCDwAgAAAAAAAAADau4cGNxRZEfNZJzG3vk2iOmVVdbwbad1uvRa7VDMj8
```

Listing 5.4: The *Base64* `Unique` representation of a block address

factory

The *factory* package provides classes for the dynamic reconstruction of serialised objects. By relying on the factory, one can receive a serialised item and reconstruct the associated object without knowledge of its original type.

cryptography

The *cryptography* package provides a set of classes and methods for performing cryptographic operations on any serialisable item. The functionalities provided range from hash functions via the `OneWay` class to symmetric and asymmetric cryptosystems through the `SecretKey`, `KeyPair`, `PublicKey`, `PrivateKey` classes, among others.

concurrency

The *concurrency* package embeds common mechanisms such as `Mutex`, `Condition`, `Semaphore`, `Thread`. Interestingly though, these functionalities are seldom used because *Infinit* has been designed to rely on the event [Maz01, LCG07] programming paradigm, as opposed to the multithreaded model.

The multithreaded paradigm relies on multiple threads operating in parallel and synchronising whenever necessary. On the other hand, the event paradigm relies on components processing events as they occur, potentially generating other events which, in turn, would be processed whenever possible. Research [AHT+02] showed that, although both models suffer pitfalls, the best of both worlds is achievable through automatic stack management.

The event programming paradigm is well-known for simplifying concurrency while reducing opportunities for race conditions and deadlocks. However, the model also

implies the event process' local statelessness. Indeed, should an event process depend on some other inputs, an event would be generated, leading to another computation and so on. Therefore, although very powerful, event-driven programming can become cumbersome as one must manually store information in the global state until an event is received in order to carry on a previously started logical unit of computation. *Adya et al.* [AHT+02] showed that a programming style exists which benefits from the event-driven paradigm's advantages without the burden of manual state management.

The *Elle* library incorporates such a continuation [MI09] mechanism through the *Fiber* class. A *fiber* is a lightweight thread of execution as it solely consists of a stack and a set of registers. Although fibers implement the event paradigm, such objects can also block while maintaining a local state. Whenever a fiber stops, another waiting fiber is scheduled by switching back on its stack as well as restoring its set of registers. Once the event necessary for the blocked fiber to continue is received, it can be re-scheduled, hence restoring its local state. The notion of fiber is extremely powerful because developers do not have to change their habits regarding state management. *Listing 5.5* provides an example of fibers requiring another resource in order to continue. Noteworthy is that both fibers, represented by the functions `Fiber1()` and `Fiber2()`, evolve within their own environment such that the value of the local variable `i` is maintained during the entire fiber's lifespan.

```
1   Timer               Timer1;
2   Timer               Timer2;

4   Resource            ResourceA;
5   Resource            ResourceB;

7   Status              Main(const Natural32                    argc,
8                            const Character*                   argv[])
9   {
10    Callback<>        fiber1(&Fiber1);
11    Callback<>        fiber2(&Fiber2);

13    // create and start the timer1, launching the fiber1.
14    if (Timer1.Create(Timer::ModeSingle, fiber1) == StatusError)
15      escape('unable to create the timer');

17    Timer1.Start(100);

19    // create and start the timer2, launching the fiber2.
20    if (Timer2.Create(Timer::ModeSingle, fiber2) == StatusError)
21      escape('unable to create the timer');

23    Timer2.Start(1000);
```

```
25        return (StatusOk);
26      }

28      Status              Fiber1()
29      {
30        Natural32         i = 42;

32        // wait for ResourceA.
33        if (Fiber::Wait(&ResourceA) == StatusError)
34          escape('unable to wait for the resource');

36        // awaken ResourceB.
37        if (Fiber::Awaken(&ResourceB) == StatusError)
38          escape('unable to awaken the resource');

40        return (StatusOk);
41      }

43      Status              Fiber2()
44      {
45        Natural32         i = 21;

47        // awaken ResourceA.
48        if (Fiber::Awaken(&ResourceA) == StatusError)
49          escape('unable to awaken the resource');

51        // wait for ResourceB.
52        if (Fiber::Wait(&ResourceB) == StatusError)
53          escape('unable to wait for the resource');

55        return (StatusOk);
56      }
```

Listing 5.5: An illustration of fibers

network

The *network* package contains high-level network functionalities from local synchronous communication mechanisms via the `Door` and `Lane` classes to remote asynchronous communication routines through `Slot`, `Gate` and `Bridge`.

Noteworthy is that every network message is identified by a `Tag` which specifies the type of the message. The `Tag` is included in a `Header` which is followed by the actual data. The `Header` also includes an `Event` number which can be used to link a response to a previously sent request for instance.

The package is especially interesting for developing networking components because

of its template-based classes and methods. As such, a component willing to send and/or receive messages must declare the associated types. This process can easily be achieved through the macro-functions `outward()` and `inward()`. *Listing 5.6* illustrates the *Agent*'s message definitions, such definitions being usually located in a file referred to as the component's *manifest*.

```
1     inward(agent::TagDecrypt,
2            parameters(elle::Code));
3     outward(agent::TagDecrypted,
4             parameters(elle::Clear));

6     inward(agent::TagSign,
7            parameters(elle::Plain));
8     outward(agent::TagSigned,
9             parameters(elle::Signature));
```

Listing 5.6: The message definition process

In addition, a component willing to receive a message must register the tag corresponding to the message of interest and associate it with the callback to trigger should such a message be received. *Listing 5.7* illustrates this straightforward process for the *Agent*'s messages defined above.

```
1     elle::Callback<const elle::Code>          decrypt(&Agent::Decrypt);
2     elle::Callback<const elle::Plain>         sign(&Agent::Sign);

4     // register the decrypt message.
5     if (elle::Network::Register<TagDecrypt>(decrypt) == elle::StatusError)
6       escape('unable to register the decrypt callback');

8     // register the sign message.
9     if (elle::Network::Register<TagSign>(sign) == elle::StatusError)
10      escape('unable to register the sign callback');
```

Listing 5.7: The message registration process

Noteworthy is that whenever a message is received, a fiber is spawned in which the associated callback is triggered. Therefore, the message handler can behave according to the fiber's specifics, such as waiting for a resource or spawning a subfiber for instance.

util

The *util* package provides miscellaneous functionalities including format classes such as `Base64` and `Hexadecimal` but also time manipulation through `Time`, arguments parser via the `Parser` class *etc.*

5.2.2 Lune

Lune is a small library which provides functionalities for manipulating locally stored information such as the user's key pairs, the addresses of the multiple *Infinit* networks the user's has been authorised to connect to *etc.*

5.2.3 PIG

PIG (POSIX/Infinit Gateway) is an application which, through *FUSE* [FUS], receives every system call related to an *Infinit* file system. The objective of *PIG* is to translate these *POSIX* system calls into requests complying with *Etoile*'s *API (Application Programming Interface)*.

Listing 5.8 illustrates *PIG*'s internals through the `rmdir()` *FUSE* call. The function starts by loading both the parent and target directories, hence retrieving two context identifiers. The directory entry pointing to the target sub-directory is then deleted. Finally, the modifications on the parent directory are commited while the target sub-directory is destroyed.

```
1   int                     PIG::Rmdir(const char*                  path)
2   {
3     etoile::path::Slice         name;
4     etoile::path::Way           child(path);
5     etoile::path::Way           parent(child, name);
6     etoile::context::Identifier directory;
7     etoile::context::Identifier subdirectory;
8
9     // load the directory.
10    if (PIG::Channel.Call(
11          elle::Inputs<etoile::TagDirectoryLoad>(parent),
12          elle::Outputs<etoile::TagIdentifier>(directory)) ==
13        elle::StatusError)
14      error(ENOENT);
15
16    // load the subdirectory.
17    if (PIG::Channel.Call(
18          elle::Inputs<etoile::TagDirectoryLoad>(child),
19          elle::Outputs<etoile::TagIdentifier>(subdirectory)) ==
20        elle::StatusError)
21      error(ENOENT);
22
23    // remove the entry.
24    if (PIG::Channel.Call(
25          elle::Inputs<etoile::TagDirectoryRemove>(directory, name),
26          elle::Outputs<etoile::TagOk>()) == elle::StatusError)
27      error(EACCES);
```

```
29        // store the directory.
30        if (PIG::Channel.Call(
31              elle::Inputs<etoile::TagDirectoryStore>(directory),
32              elle::Outputs<etoile::TagOk>()) == elle::StatusError)
33          error(EINTR);

35        // destroy the subdirectory.
36        if (PIG::Channel.Call(
37              elle::Inputs<etoile::TagDirectoryDestroy>(subdirectory),
38              elle::Outputs<etoile::TagOk>()) == elle::StatusError)
39          error(EINTR);

41        return (0);
42      }
```

Listing 5.8: The *PIG*'s `rmdir()` *POSIX* system call

Note that although most *FUSE* calls are converted into *Infinit* requests, some are ignored, such as `link()`, while the semantics of others are slightly modified in order to comply with *Infinit*. For instance the `getattr()` *FUSE* call is invoked in order to retrieve information on the object referenced by the given path. Such information ranges from the object's mode, to the owner's *UID* and *GID*, to the data size and so on. Unfortunately, the system cannot translate an *Infinit* user/group identifier into a *UNIX* *UID*/*GUID*, respectively. The system has to be provided with a set of pre-defined mappings between *Infinit* and *UNIX* identifiers. The local file `$HOME/.infinit/map.asct` provides such a mapping. However, should the system fail to translate an *Infinit* identifier, *PIG* would rely on a special entity referred to as *somebody*. The *somebody* *UNIX* user and group are therefore attributed to any file system object whose owner identity has not been linked to a *UNIX* entity.

5.2.4 Agent

The *Agent* component aims at performing cryptographic operations on behalf of a specific user. This component has been introduced in order to isolate the sensitve user's key pair from other potentially more vulnerable components. Noteworthy is that this data isolation process is similarly used in *SSH (Secure Shell)* through its so-called *SSH* agent.

Whenever *Etoile* receives a request that requires performing a cryptographic operation, the component contacts the user's *Agent* and requests the operation to be performed. That way, the user's keys never leave its *Agent* whose source code, being extremely small, is easier to maintain and secure than *Etoile* or *Hole* for instance.

The *Agent* component starts by connecting and identifying itself to the *Etoile* component. The *Etoile* component then challenges the *Agent* regarding the identity of the user by sending a phrase encrypted with the user's public key. The *Agent* decrypts the phrase and sends back a hash of the phrase, hence proving that it legitimally operates on behalf of the claimed user. Besides, the phrase is stored in the local file `$HOME/.infinit/$USER.phr` so that the user's applications, including *PIG*, can authenticate to *Etoile* by providing this phrase.

As discussed in *Section 5.2.1*, the *Agent* exports an interface composed of functionalities for decrypting and signing data with the user's private key. *Listings 5.9 & 5.10* provide the source code for these operations.

```
1   elle::Status          Agent::Decrypt(const elle::Code&        code)
2   {
3     elle::Clear         clear;

5     // perform the cryptographic operation.
6     if (Agent::Pair.k.Decrypt(code, clear) == elle::StatusError)
7       escape('unable to perform the decryption');

9     // reply to the caller.
10    if (Agent::Channel->Reply(
11          elle::Inputs<TagDecrypted>(clear)) == elle::StatusError)
12      escape('unable to reply to the caller');

14    return (StatusOk);
15  }
```

Listing 5.9: The `Agent::Decrypt()` method

```
1   elle::Status          Agent::Sign(const elle::Plain&          plain)
2   {
3     elle::Signature     signature;

5     // perform the cryptographic operation.
6     if (Agent::Pair.k.Sign(plain, signature) == elle::StatusError)
7       escape('unable to perform the signature');

9     // reply to the caller.
10    if (Agent::Channel->Reply(
11          elle::Inputs<TagSigned>(signature)) == elle::StatusError)
12      escape('unable to reply to the caller');

14    return (StatusOk);
15  }
```

Listing 5.10: The `Agent::Sign()` method

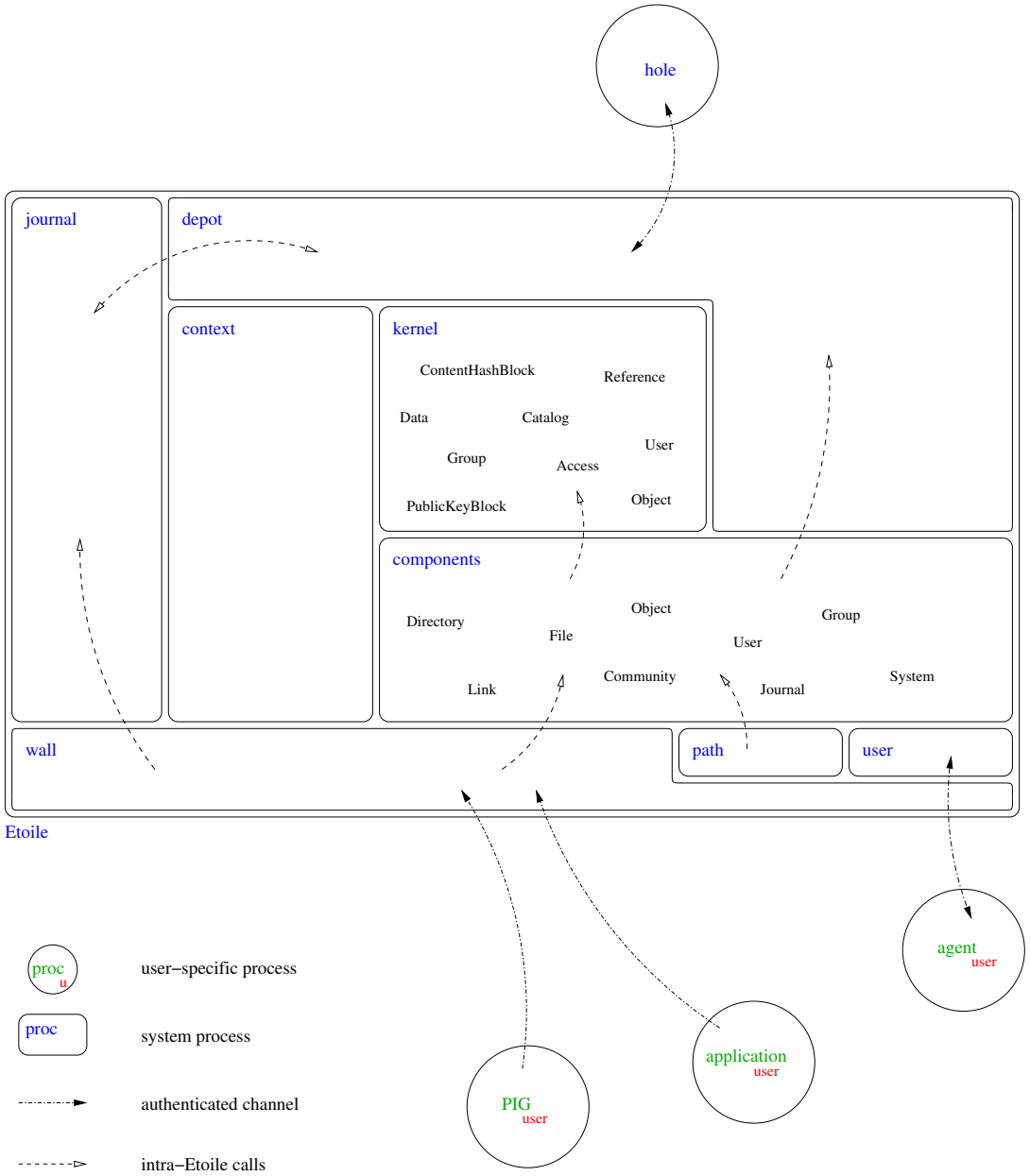


Figure 5.6: The internals of the *Etoile* component

5.2.5 Etoile

The *Etoile* component implements the *Infinit* file system *API*. Following *Infinit*'s architecture and as shown in *Figure 5.6*, *Etoile* is partitioned into logical units such as *wall*, *path*, *context*, *kernel etc.* which are discussed below.

wall

The *wall* unit exports *Etoile*'s interface such that every incoming message generates an event triggering a method located in this unit. *Listing 5.11* provides some of

Etoile's message definitions while *Listing 5.12* illustrates the related event handlers. The reader will notice that these definitions correspond to the messages sent by *PIG* in *Listing 5.8*.

Whenever a request is received, *wall* retrieves the context associated with the given context identifier. The request is then passed to the *components* unit. Finally, a response is sent back to the requesting client.

```
1     // directory
2     inward(etoile::TagDirectoryCreate,
3            parameters());
4     inward(etoile::TagDirectoryLoad,
5            parameters(etoile::path::Way));
6     inward(etoile::TagDirectoryAdd,
7            parameters(etoile::context::Identifier,
8                       etoile::path::Slice,
9                       etoile::context::Identifier));
10    inward(etoile::TagDirectoryLookup,
11           parameters(etoile::context::Identifier,
12                      etoile::path::Slice));
13    outward(etoile::TagDirectoryEntry,
14            parameters(etoile::kernel::Entry));
15    inward(etoile::TagDirectoryConsult,
16           parameters(etoile::context::Identifier,
17                      etoile::kernel::Index,
18                      etoile::kernel::Size));
19    outward(etoile::TagDirectoryRange,
20            parameters(etoile::kernel::Range<etoile::kernel::Entry>));
21    inward(etoile::TagDirectoryRename,
22           parameters(etoile::context::Identifier,
23                      etoile::path::Slice,
24                      etoile::path::Slice));
25    inward(etoile::TagDirectoryRemove,
26           parameters(etoile::context::Identifier,
27                      etoile::path::Slice));
28    inward(etoile::TagDirectoryDiscard,
29           parameters(etoile::context::Identifier));
30    inward(etoile::TagDirectoryStore,
31         parameters(etoile::context::Identifier));
32    inward(etoile::TagDirectoryDestroy,
33           parameters(etoile::context::Identifier));
```

Listing 5.11: *Etoile*'s *wall* message definitions for directory objects

```
1     class Directory
2     {
3     public:
4       static elle::Status       Create();
```

```
5       static elle::Status       Load(const path::Way&);
6       static elle::Status       Lock(const context::Identifier&);
7       static elle::Status       Release(const context::Identifier&);
8       static elle::Status       Add(const context::Identifier&,
9                                     const path::Slice&,
10                                    const context::Identifier&);
11      static elle::Status       Lookup(const context::Identifier&,
12                                       const path::Slice&);
13      static elle::Status       Consult(const context::Identifier&,
14                                        const kernel::Index&,
15                                        const kernel::Size&);
16      static elle::Status       Rename(const context::Identifier&,
17                                       const path::Slice&,
18                                       const path::Slice&);
19      static elle::Status       Remove(const context::Identifier&,
20                                       const path::Slice&);
21      static elle::Status       Discard(const context::Identifier&);
22      static elle::Status       Store(const context::Identifier&);
23      static elle::Status       Destroy(const context::Identifier&);
24  };
```

Listing 5.12: *Etoile*'s *wall* handler definitions for directory objects

context

Depending on the request, a *context* is either created, loaded, stored or discarded. For example, a call such as `Directory::Load()` creates a context while `Directory::Remove()` retrieves an existing context according to the given `Identifier`. The notion of context has been introduced in order to optimise the system's performance by delaying expensive operations. As detailed in *Chapter 4*, an *Object* logical block contains several signatures. These signatures are computed in order to seal the object before being stored in the peer-to-peer network. However, considering a file system devoid of the notion of context, signatures would be re-computed following every call to the `write()` *POSIX* function. One can easily imagine many scenarios such as copying files leading to a large number of signatures being unnecessarily computed. Contexts, on the other hand, enable an application to perform a set of operations on an object before commiting the modifications, in which case the object is sealed and finally stored in the distributed hash table. The *context* unit contains functionalities for manipulating such contexts.

components

The *components* unit provides a very similar interface to the *wall*'s, carrying on operations on a given context. The objective of the *components* unit is to maintain

consistency between the blocks composing an object. Indeed, while the *kernel* unit provides functionalities at the block level, the *components* unit ensures that, should the *Access* or *Contents* blocks be modified for instance, the *Object* is updated accordingly. *Listing 5.13* illustrates the `Remove()` method for a directory object. This method takes the context of the directory object along with the name of the entry to remove. First, the requesting client's rights are determined and checked. Then, the contents is opened by potentially fetching additional blocks. Finally, the *Contents* and *Catalog* blocks are modified by requesting the *kernel* unit.

```
1     elle::Status        Directory::Remove(context::Directory*   context,
2                                           const path::Slice&    name)
3     {
4       // compute the current user's rights on the context.
5       if (Rights::Determine(context) == elle::StatusError)
6         escape('unable to determine the rights');
7
8       // check the user's rights according to the operation.
9       if (!(context->rights->record.permissions & kernel::PermissionWrite))
10        escape('unable to perform the operation without the permission');
11
12      // open the contents.
13      if (Contents::Open(context) == elle::StatusError)
14        escape('unable to open the contents');
15
16      // remove the entry from the directory contents.
17      if (context->contents->content->Remove(name) == elle::StatusError)
18        escape('unable to remove the directory entry');
19
20      return (StatusOk);
21    }
```

Listing 5.13: The *components* unit's `Directory::Remove()` method

path

The *path* unit provides methods for manipulating paths such as resolving a path into an object's address.

user

The *user* unit contains information regarding the clients connected to *Etoile*. A client is composed of an agent and a set of applications such as *PIG*. Whenever the connection with the client's *Agent* is broken, all the applications are notified and their connection discarded.

kernel

The *kernel* unit contains the internal representations of the fundamental blocks. The unit includes classes such as `Object`, `Access`, `Contents`, `Data`, `Reference`, `User`, `Group`, `ContentHashBlock` among others. *Listing 5.14* depicts the internal representation of an *Object* logical block through its `Object` class. The reader will notice that this structure closely resembles the *OKB*-based *Object* logical block depicted on *Figure 4.6*.

```
1   class Object:
2     public OwnerKeyBlock
3   {
4     Author              author;

6     struct
7     {
8       struct
9       {
10        Permissions     permissions;
11        Token           token;
12      }                 owner;

14      Genre             genre;
15      elle::Time        stamp;

17      Attributes        attributes;

19      hole::Address     access;

21      Version           version;
22      elle::Signature   signature;
23    }                   meta;

25    struct
26    {
27      hole::Address     contents;

29      Size              size;
30      elle::Time        stamp;

32      elle::Digest      fingerprint;

34      Version           version;
35      elle::Signature   signature;
36    }                   data;
37  };
```

Listing 5.14: The `Object` class

depot

The *depot* unit provides an abstraction for storing and retrieving blocks, independently of their location. Indeed, although the *depot* may, most of the time, rely on the *Hole* component in order to retrieve data from the underlying distributed hash table, the block could also lie in one of *Etoile*'s caches, either in the *depot*'s cache, referred to as the *repository*, or in the *journal*'s. Note that a block located in the *repository* may actually be stored in either the *cache i.e.* in main memory or in the *reserve i.e.* on the local hard disk. Therefore, whenever a block is requested by the *components* unit, the *depot* is invoked. The unit then sequentially inspects the *journal*, the *cache*, the *reserve* and, at last, sends a request to the *Hole* component.

journal

Whenever the modifications applied to a context are requested to be commited, the context is pushed into the *journal*. As discussed throughout *Chapter 4*, updates are delayed in order to optimise the system's performances since blocks could be subsequently modified or even destroyed. Then, on a periodic basis, the blocks are stored in the distributed hash table by requesting the *depot*'s `Put()` method.

5.2.6 Hole

The *Hole* component provides an abstraction for storing, retrieving and deleting a `Block` associated with an `Address`. The interface exported by the component is composed of the methods listed in *Listing 5.15*. Note that this interface closely resembles the one defined in *Section 3.4*.

```
1     class Hole
2     {
3     public:
4       static elle::Status       Put(const Address&,
5                                     const Block*);
6       static elle::Status       Get(const Address&,
7                                     Block*&);
8       static elle::Status       Erase(const Address&);
9     };
```

Listing 5.15: The `Hole` component's interface

Noteworthy is that an `Address` contains a header composed of *(i)* a *family* representing the physical block *(ii)* a *component* representing the logical block and *(iii)* the hash of the creating user's public key. This particular construct prevents different blocks' addresses from conflicting.

Since this interface is common [DZD+03] to most *DHT*s, the implementation can easily be changed, even at run-time. The *Infinit* prototype is currently composed of the *Hole* implementations discussed next.

local

The *local* implementation stores the data blocks locally. This implementation, though benefiting from extreme simplicity, obviously lacks fundamental properties such as availability, durability, scalability *etc.*

remote

The *remote* implementation stores the data blocks on a unique remote server. As for *local*, the *remote* implementation suffers from fundamental limitations. However both these implementations are useful regarding debugging.

kool

Finally, the *kool* implementation stores blocks in a distributed hash table, hence complying with the properties defined in *Chapter 3*.

As the mobility property defined in *Chapter 3* suggests, devices lacking the resources to contribute or even maintain the network's state should be able to configure the distributed hash table accordingly. Structured overlay networks have been designed with scalability in mind. For instance, *Chord* [SMK+01] performs lookups in $O\big(log(\eta)\big)$ hops while nodes are required to maintain a state composed of $O\big(log(\eta)\big)$ entries, where η represents the number of nodes. Unfortunately, a node could find itself in a position where the network state it must maintain exceeds its storage capacity.

kool follows the *Kelips* [GBL+03] design but extends it so that the *DHT* can be configured by specifying the degree of partitioning δ. This characteristic implies that, as δ increases, the routing path is lengthened while the network state to be maintained on every node is reduced. Therefore, while the routing complexity remains $O\big(1\big)$, the state complexity becomes $O\big(\sqrt[\delta]{\eta}\big)$.

For instance, $kool^2$ represents a *Kelips* network in which nodes are assigned to groups, $kool^3$ relies on cube roots, hence dividing the space into a two-level hierarchy while $kool^4$ goes one step further and partitions the identifier space into three layers.

By relying on such a flexible *DHT*, *Infinit* can easily be set up on a variety of exotic devices. Mobile phones for instance would probably make use of $kool^4$ in order to

reduce the memory fingerprint while a desktop computer could afford to trade off memory consumption for faster lookups through $kool^2$ or even $kool^1$.

Although such configurations could be applied to other overlay networks such as *Chord* [SMK+01], the *Kelips* [GBL+03] design has been chosen for its simplicity.

Table 5.1 summarises the relations between the several configuration factors given a degree of partitioning δ, a network designed to support η nodes and a set of connected nodes ζ. The *cardinality* represents the size of every group, the number of groups in every level of the hierarchy as well as the hierarchy depth. The *neighbourhood* parameter specifies the number of connected nodes populating every group. This equation assumes the use of a consistent hash function for attributing nodes to groups in a uniform manner. Finally, the *connectivity* indicates the number of links a node must maintain *(i)* with every other node in its group *i.e.* the neighbourhood and *(ii)* with every other group belonging to the node's hierarchical level plus every super-group in the hierarchy upper-level plus every super-super-group in the upper-upper-level and so on. Note that every link to groups, upper-groups, upper-upper-groups *etc.* are made redundant according to the ratio γ_d depending on the degree d. However, for the sake of simplicity the following assumes that $\gamma = \gamma_d$ for every degree d.

Cardinality	$\kappa = \sqrt[\delta]{\eta}$
Neighbourhood	$\phi = \dfrac{\zeta}{\kappa^{\delta-1}}$
Connectivity	$\lambda = \phi - 1 + \sum_{d=1}^{\delta-1} \Big(\gamma_d \big(\kappa - 1 \big) \Big)$

Table 5.1: *kool* parameters

Figure 5.7 depicts a network illustrating the relations between the parameters detailed above. This example shows how every group is populated by an average of $\phi = 2$ nodes while every node maintains a total of $\lambda = 5$ links with the other groups and upper-groups.

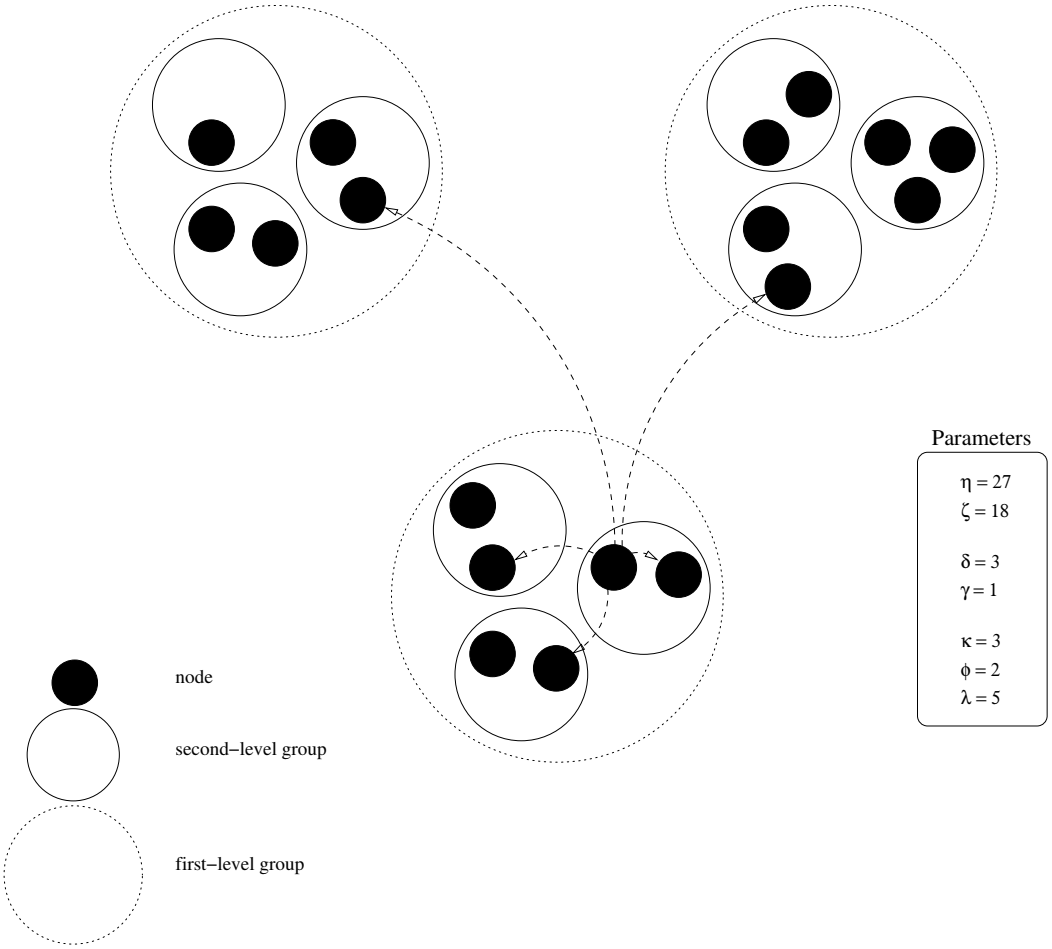


Figure 5.7: An example of a $kool^3$ network

Table 5.2 provides formulas for computing the number of hops as well as the size of the network state nodes are expected to maintain. Note that the state is assumed to be composed of a table of 30-byte entries containing the target node's *IP (Internet Protocol)* address.

Hops	$\delta - 1$
State	$\lambda \times 30B$

Table 5.2: *kool* formulas

Finally, *Table 5.3* compares three different *kool* configurations capable of supporting $\eta = 10^{10}$ nodes. The network is assumed to be populated by $\zeta = 10^9$ nodes while the connectivity redundancy ratio is set to $\gamma = 3$, hence improving resilience. This study shows that the state can be reduced from $8.86MB$ for $kool^2$ to $384.72KB$ for $kool^3$, representing a gain of 95.7% at the expense of a single additional hop.

Network	Configuration	Parameters	Hops	State
$\eta = 10^{10}$ $\zeta = 10^9$ $\gamma = 3$	$kool^2$ $\delta = 2$	$\kappa = 10^5$ $\phi = 10^4$ $\lambda = 309996$	1	$8.86MB$
	$kool^3$ $\delta = 3$	$\kappa = 2154$ $\phi = 215$ $\lambda = 13132$	2	$384.72KB$
	$kool^4$ $\delta = 4$	$\kappa = 316$ $\phi = 31$ $\lambda = 2865$	3	$83.95KB$

Table 5.3: Comparison of the *kool* configurations

Although *Kelips* would incur a high background communication overhead to keep a $8.86MB$ state up-to-date, *kool* can be configured to trade off one additional hop against a state reduction. This flexibility appears extremely useful since resource-limited devices such as mobile phones should be able to maintain a $83.95KB$ state.

Note that an overlay network such as *Chord* [SMK+01] could also benefit from such flexible configurations by modifying the base parameter for instance.

⋆ ⋆
⋆

The presented prototype follows the *UFS* representation consisting in expressing every file, directory and link through an *inode*. Given the decentralised nature of the *Infinit* peer-to-peer file system, *inodes* are not stored in the kernel or on a dedicated server but rather in the underlying distributed hash table; *inodes* are represented by the *Object* logical block which has been detailed throughout *Chapter 4*.

Regarding the actual implementation of the *Infinit* prototype, the system has been broken down into independent units which communicate by sending messages, also referred to as *IPC (Inter-Process Communication)* [DMM08]. Finally, the *Hole* component which implements the storage layer interface offers several network implementations which can be used according to the environment: topology, constraints *etc.*

Chapter 6

Evaluation

The administration scheme presented in *Section 4.2* enables users to request special tasks. However, such operations may take several days to complete, depending on the availability and commitment of the users contributing to the system. Unfortunately, this particular aspect of the system cannot be evaluated in an unrealistic environment; a well-established production environment with hundreds of users is required to perform a long term analysis, especially regarding the impact of the users' connectivity as well as the concurrency of administrative requests.

Likewise, the access control scheme is particularly interesting because of its extreme flexibility. Unfortunately, this expressivity comes at the expense of an increased connectivity requirement. Although it has been shown in *Section 4.1.7* that such a connectivity was realistically achievable, evaluating its impact would, once again, require a production environment populated by hundreds of users sharing files, creating and managing groups and so forth.

Since evaluating the qualitativeness of the file system through a simulation is inconceivable because reliance on human actions for validating administration requests, for example, and since setting up a production environment may take years to achieve, the evaluation of the *Infinit* design and implementation has been performed in several complementary environments.

This chapter therefore discusses the details of this evaluation. Firstly, aspects of the underlying overlay network and distributed hash table are evaluated, such as the routing latency and the network scalability. Secondly, the peer-to-peer file system is evaluated both regarding its implementation and deployment especially considering the access control and administration mechanisms designed in *Chapter 4*.

6.1 Methodology

This section discusses the environments, benchmarks and data sets this evaluation relies upon.

6.1.1 Environments

The large-scale specificity of peer-to-peer file systems implies that experiments should be carried out in an appropriate environment *i.e.* a network populated by thousands of nodes and hundreds of users. Unfortunately, since setting up such a network is extremely complicated and time-consuming, researchers tend to rely on other techniques. This evaluation follows the common practice which consists in measuring the file system's behaviour in several complementary environments, offering a trade-off between realism and scalability.

The first environment, referred to as the *simulated* environment, makes use of the *ns-3* [NSN] discrete-event network simulator. The simulator is run on a *Linux*-based computer with a dual-core $1.8Ghz$ microprocessor and $4GB$ of *RAM (Random Access Memory)*. This environment is especially useful to simulate a large-scale network composed of several thousand nodes. Thus, both the evaluations of the overlay network and the distributed hash table take place in this environment. The network topology used by these benchmarks has been generated with the *iNet Topology Generator* [iNe] in which the maximum latency between diametrically opposed nodes has been set to $300ms$.

The second environment, known as the *realistic* environment, is composed of 16 heterogeneous nodes located throughout Europe and the United States of America. This environment is used to evaluate the implementation of the *Infinit* file system, in a more realistic way, through the use of *Andrew* benchmarks.

Finally, the third environment, referred to as the *production* environment, is composed of $2,156$ heterogeneous nodes populating a campus network split into two geographic sites. This network is used on a daily basis by five schools for approximately $6,000$ users including students, teachers, staff *etc.* This environment is used to evaluate the behaviour of the *Infinit* file system in a production environment in order to validate the assumptions made regarding the connectivity requirements for both the access control and administration schemes.

6.1.2 Benchmarks

This section presents the three types of benchmarks carried out on the system along with their processing methods and data sets.

Firstly, the evaluation focuses on the overlay network and distributed hash table components. The experiments carried out on the *kool* implementation aim at evaluating some characteristics of the underlying network such as the latency of the routing requests. In order to perform such experiments, the *simulated* environment is set up for hosting several thousand nodes. Then, a benchmark composed of a pre-generated set of requests is run, hence stressing the network's behaviour.

Secondly, the peer-to-peer file system implementation is evaluated and compared with *NFS* [Osa88] through the *Andrew* [HKM+88b] benchmark. This methodology makes *Infinit* comparable with many other file systems which have been evaluated in very similar conditions, including *OceanStore* [KBC+00], *Ivy* [MMGC02], *Pangaea* [SKKM02], *FARSITE* [ABC+02] and *Pastis* [mBPS05].

The *Andrew* [HKM+88b] benchmark aims at individually evaluating specific aspects of the file system. This benchmark is composed of several phases, each one dealing with a particular file system operation such as copying directories or creating files through a compilation process. The source code of the *Andrew* benchmark used throughout this evaluation is given in *Listing 6.1*.

```
1   #! /bin/sh

3   Prepare()
4   {
5     echo ``---[ Prepare''
6     cd ``${from}/''
7     directories=$(find ./ -type d)
8     time                                                    \
9       (for directory in ${directories} ; do
10        mkdir ``${to}/${directory}''
11      done) >>andrew.log
12  }

14  Copy()
15  {
16    echo ``---[ Copy''
17    cd ``${from}/''
18    files=$(find ./ -type f -or -type l)
19    time                                                    \
20      (for file in ${files} ; do
21        cp -P ``${file}'' ``${to}/${file}''
22        if [ ${?} -ne 0 ] ; then exit 1 ; fi
23      done) >>andrew.log
24  }

26  List()
27  {
28    echo ``---[ List''
```

```
29        time                                                          \
30          (ls -Rla ``${to}'') >>andrew.log
31      }

33      Search()
34      {
35        echo ``---[ Search''
36        time                                                          \
37          (grep -R ``teton'' ``${to}'') >>andrew.log
38      }

40      Compile()
41      {
42        echo ``---[ Compile''
43        cd ``${to}/''
44        time                                                          \
45          (./configure && make) >>andrew.log
46      }

48      if [ ${#} -ne 2 ] ; then
49        echo ``[usage] andrew.sh {from} {to}''
50        exit 0
51      fi

53      cd ``${1}''
54      from=''${PWD}''
55      cd ``${OLDPWD}''

57      cd ``${2}''
58      to=''${PWD}''
59      cd ``${OLDPWD}''

61      rm -f andrew.log

63      Prepare
64      Copy
65      List
66      Search
67      Compile
```

Listing 6.1: The *Andrew* benchmark

The first phase of the *Andrew* benchmark, referred to as *Prepare* in *Listing 6.1*, clones the hierarchy of directories of a given project. The second phase copies the files into the freshly created hierarchy. The *List* phase lists all the files and directories created so far, hence retrieving the attributes of every file system object. The fourth phase, known as *Search*, reads the content of every file by invoking the `grep` utility.

Finally, the fifth phase launches the `make` command, hence compiling and linking the source files.

This benchmark has been, and still is, widely used for evaluating file systems because every phase focuses on a specific aspect. For instance, the first and second phases illustrate the process of creating directories and files, respectively. Likewise, the third phase focuses on inspecting every object's metadata information while the fourth retrieves the files' content. The fifth phase however is more general as it includes all the previous operations: retrieving *Make* files' attributes, reading source files and writing object files, among others.

The *Andrew* benchmark therefore takes a project directory containing *Make* files and source files and measures the duration of every phase. In order to make this evaluation's experiments as realistic as possible, the executions of the *Andrew* benchmark are based on the project directory of *OpenSSL-1.0.0*. This project is composed of approximately 900 source files and 200 header files while the compilation process generates around 700 object files.

Thirdly, the *Infinit* file system is stressed in a *production* environment by replaying the operations recorded in a 3-week file system trace generated from a system in production. This benchmark is extremely interesting for measuring two things: the accessibility of the file system objects depending on the users' connectivity and the conflicts generated by concurrent updates.

In order to validate the assumptions defined in *Chapter 4*, the *Infinit* file system is evaluated by setting up an environment in which user and group entites have been created in order to best match the file system trace, especially when it comes to sharing and working cooperatively. The following gives some insights into the file system trace.

Table 6.1 provides general information on the file system trace. This table indicates the number of files and user accounts being active, as opposed to the total number of users and files. Indeed, since the file system trace covers a 3-week period, only the users and files recorded in the trace are considered.

Number of users	$5,932$
Number of files	$1,339,776,000$

Table 6.1: General information regarding the users and files

Noteworthy is that the *Infinit* file system set up for this evaluation relies on the campus' topology which implies that the nodes do not belong to the users as is common in open peer-to-peer networks. Instead, an instance of *Hole*, set up by the system administrators, runs on every computer so that every user must launch her

own *Infinit* client in order to access her files. This specificity mainly implies two things regarding the design presented in *Chapter 4*. First, the nodes populating the network are highly available since always running; this is part of the campus policy. Therefore, the peer-to-peer network does not suffer from churn. Second, since the nodes do not belong to the users, the users' *Infinit* instance is running as long as the user is logged in the campus network. This is very different from an open peer-to-peer network in which a user could have multiple *Infinit* instances running: one on her smartphone, another on her desktop computer and so on. This particularity implies that contacting a user—in order to request an operation or provide a key for instance—is far more complicated as the users' connectivity is impacted.

Average Node Availability	87.4%
Average Daily User Uptime	5 hours, 48 minutes and 33 seconds

Table 6.2: General information regarding the nodes and users connectivity

Table 6.2 provides information regarding the average node availability and user uptime. The user uptime represents the period of time during which the user is logged in the campus network, daily. Note that the uptime is extremely high because only the users repreented in the file system trace are taken into account. This period directly impacts the availability of a user's *Infinit* instance for responding to other users' requests.

The following discusses the interactions betweens user and group entities by analysing the sharing properties of files, directories *etc.*, both in reading and writing.

First, the sharing distribution is studied shedding light on how and with whom users share objects in reading. *Figure 6.1* shows with how many users file system objects are being shared. This study confirms the fact that most objects, *i.e.* 79.9%, are kept private in which case, in an *Infinit* file system, no *Access* block would be referenced. This figure also shows that users tend to share with many users, up to twelve, with an unexpected peak for sixteen readers, though most shared objects, *i.e.* 99.8%, are shared with a single user. Note that the system files, which are accessible by any user, have been ignored from this evaluation.

The second analysis focuses on file system objects that are writable by multiple users, implying some sort of a cooperative behaviour. *Figure 6.2* shows that, as expected, the number of such *"cooperative"* objects is lower than the number of shared ones. It is however interesting to notice that these measures indicate that users tend not to cooperate, or at least not to rely on file systems access control mechanisms for doing so. Indeed, a further analysis showed that many users relied upon third-party applications such as *Subversion*, *Git etc.* for cooperative work,

especially given the fact that the campus' *IT (Information Technology)* department provides functionalities for setting up such tools.

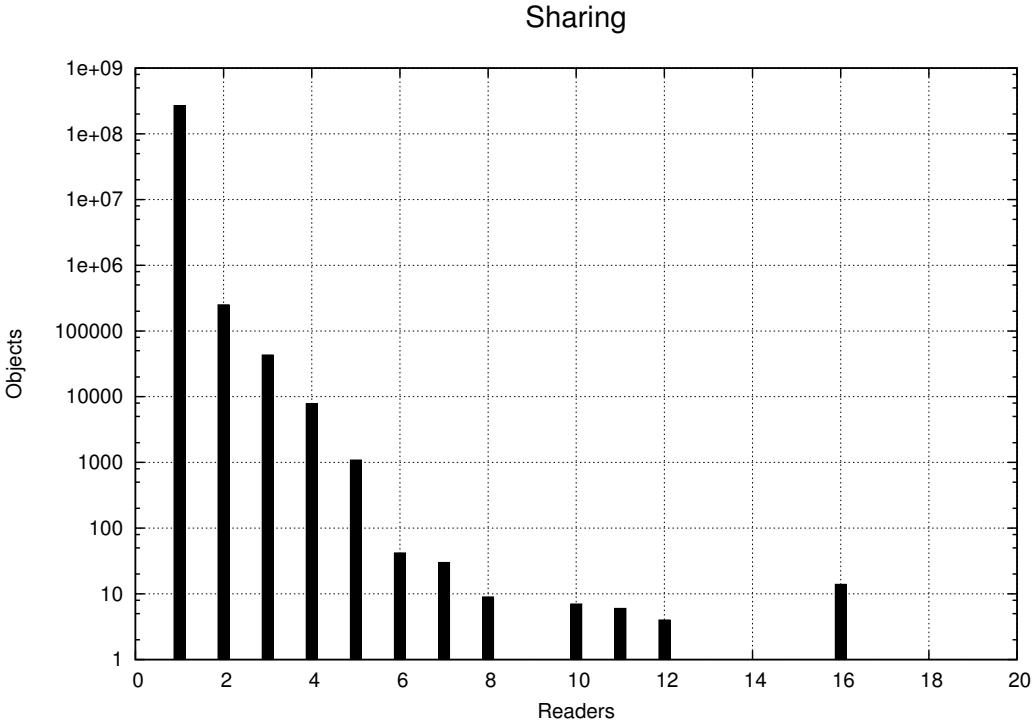


Figure 6.1: General information regarding the users' sharing behaviours

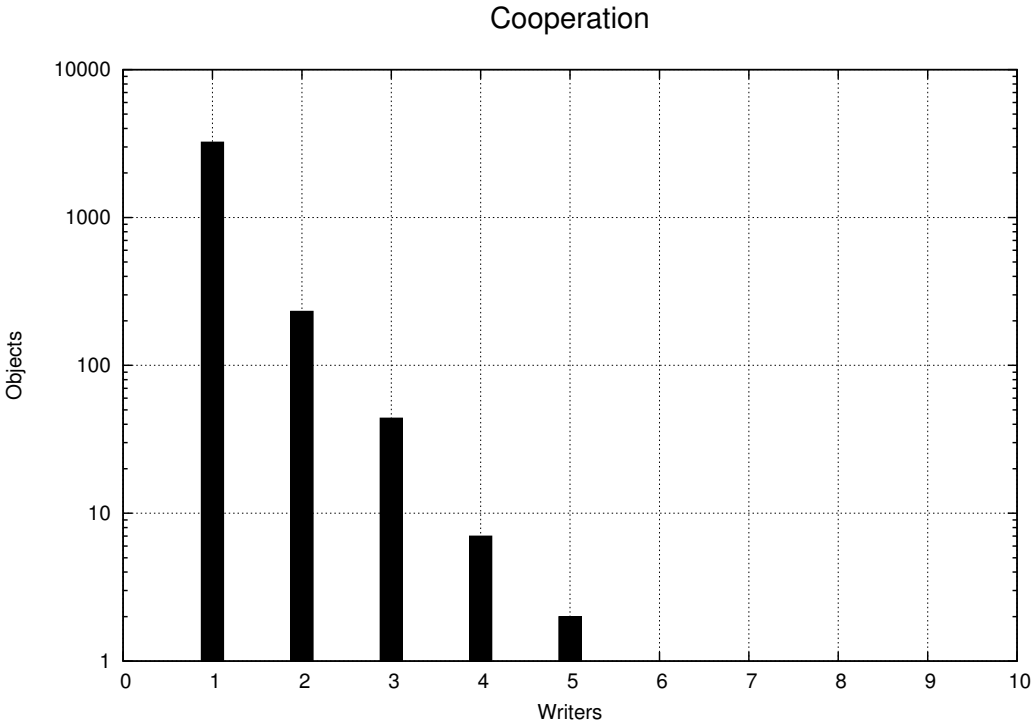


Figure 6.2: General information regarding the user's cooperative behaviours

This analysis of the file system trace provides enough information for setting up *Infinit* file system environments in which the evaluation of both the access control and administration mechanisms will take place.

6.1.3 Metrics

The primary metric this evaluation is interested in relates to the overlay network, especially regarding the latency of the routing requests. Likewise, the distributed hash table's benchmarks are intended to illustrate the complexity of the operations consisting of storing and retrieving blocks.

As a second step, the *Infinit* file system implementation will be stressed, the evaluation focusing on the execution time of the various *Andrew* benchmark phases.

Finally, the benchmarks taking place in the *production* environment concentrate on analysing the accessibility of the file system objects along with the concurrency of the updates. These metrics will be analysed through a two-phase process. First, an *Infinit* file system will be set up in order to study the access control mechanism. For this, all the file system's *Object* blocks will be built upon *OKB*s. Then, another environment will be set up in order to examine the cooperative interactions through the use of *TKB*-based objects.

Note that in order to ensure the results' validity, every benchmark is run several times.

6.2 Results

This section discusses various results, from the impact of the cryptosystems on the overall performance to the latency of the block retrieval process. Also, the *Infinit* file system's behaviour is evaluated in more realistic environments, hence assessing the impact of the access control and administration schemes designed in *Chapter 4*.

6.2.1 Overlay Network

The first set of experiments aims at validating the routing algorithm of the *kool* implementation, as presented in *Section 5.2.6*. This evaluation illustrates the performance of the `Lookup`(ι) routine in several *kool* configurations and depending on the number of nodes composing the peer-to-peer network.

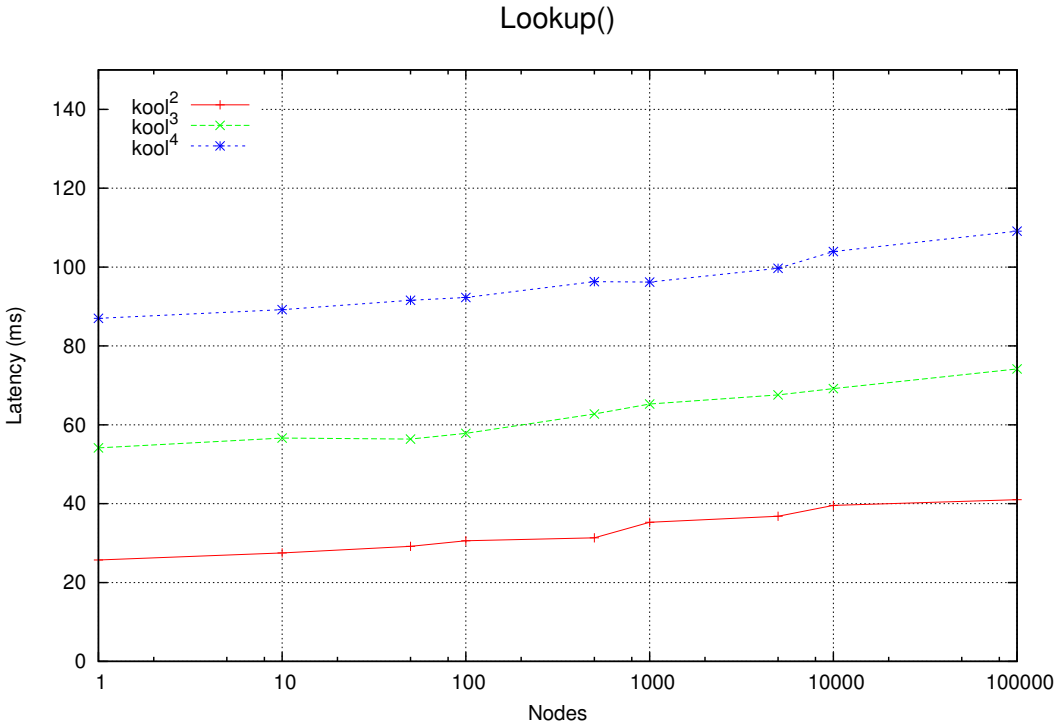


Figure 6.3: The performance of the overlay network's `Lookup`(ι) routine

The results of *Figure 6.3* confirm the design expectations of *Section 5.2.6* *i.e.* the requests' latency evolves according to the routing path's length. Let us recall that the $kool^2$ configuration performs lookups in a single hop while the $kool^3$ and $kool^4$ require two and three hops, respectively. Noteworthy is that the number of nodes has a small though noticeable impact on the routing complexity, probably because of the node failures which force the clients to re-send some messages.

6.2.2 Distributed Hash Table

The multiple physical and logical blocks defined throughout *Chapter 4* differ from one another depending on their function but also their mutability. The mutability property is especially important in regard to the performance of the underlying distributed hash table. Indeed, as explained in *Section 3.4*, the `Get`(α) and `Gather`(α) routines exhibit radically different behaviours. While the `Get`(α) method can easily identify the block but also make extensive use of caching, the `Gather`(α) method is required to contact a quorum of nodes responsible for the given block.

The following experiments concentrate on the distributed hash table, starting with the `Get`(α) routine. Note that for the sake of simplicity, the next experiments are carried out in a simulated network of $10,000$ nodes.

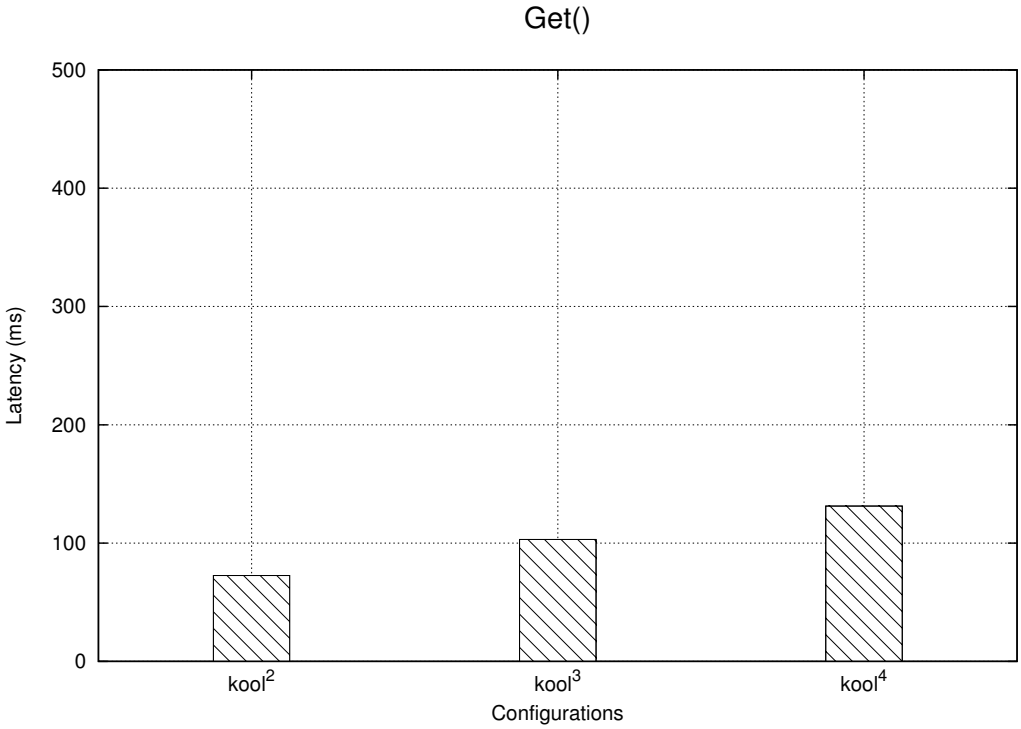


Figure 6.4: The performance of the immutable-specific `Get`(α) routine

Figure 6.4 provides the results of the experiments carried out on 4096-byte chunks of data. Let us recall that the `Get`(α) method can benefit from caching techniques because of the immutability property of the blocks associated with this routine. Note however that the cache has been disabled throughout these experiments. The results show that immutable blocks can be retrieved very efficiently from the distributed hash table. This is especially interesting since most blocks composing the file system's hierarchy are *CHB*s such as *Access*, *Data*, *Contents*, *Catalog*, *Members* *etc.*

The next set of experiments focuses on the performance of the distributed hash table regarding mutable blocks. The benchmarks consist of a set of `Gather`(α) calls involving 4096-byte mutable blocks. These experiments illustrate the behaviour of the distributed hash table under several quorum configurations identified by their fault-tolerance factor γ *i.e.* the number of Byzantine nodes such a quorum can tolerate. For instance, $\gamma = 3$ refers to a quorum configuration in which every block is replicated on 10 storage nodes while every client willing to retrieve a block must assemble a read quorum composed of 7 of those nodes. Likewise, the $\gamma = 9$ quorum configuration consists of 28 storage nodes while clients must assemble a read quorum of 19 nodes. As expected and illustrated by *Figure 6.5*, the `Gather`(α) method is several times more expensive than its `Get`(α) counterpart since several block instances must be retrieved directly from the quorum of storage nodes.

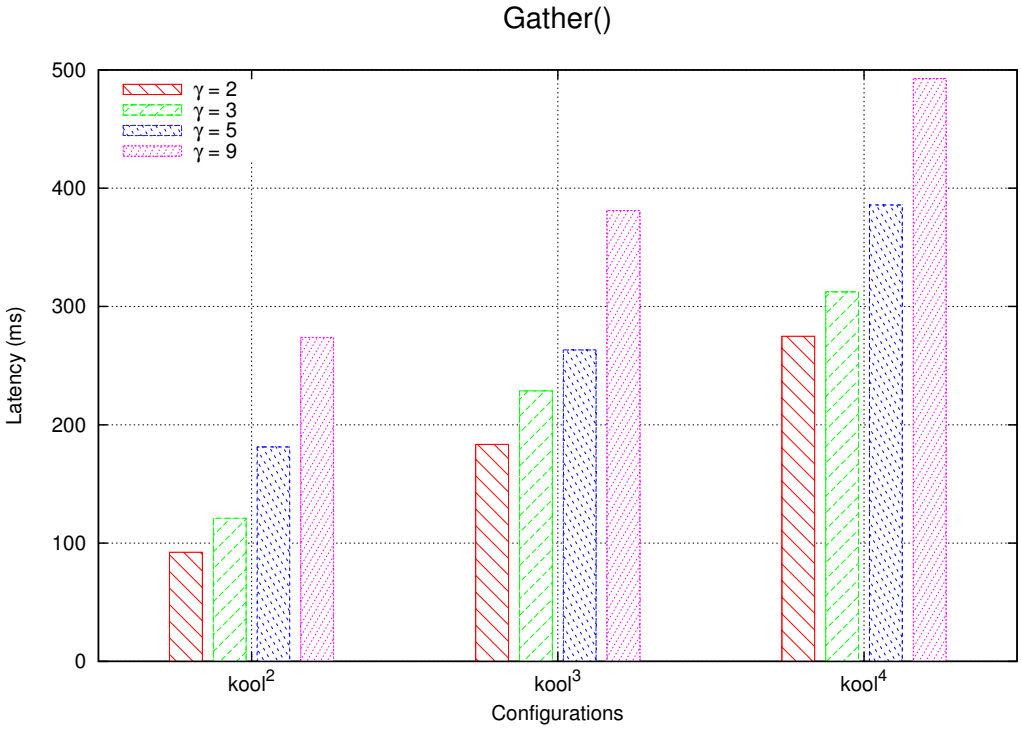


Figure 6.5: The performance of the mutable-specific `Gather`(α) routine

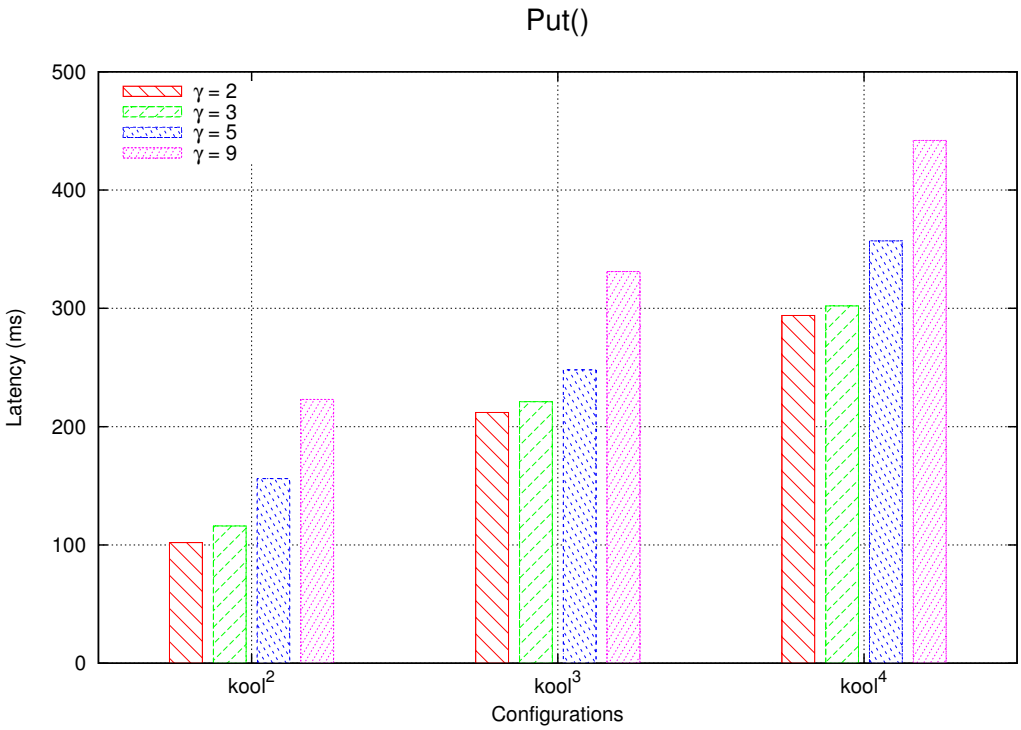


Figure 6.6: The performance of the `Put`(α, β) routine

Finally, the `Put`(α, β) routine is evaluated in a similar environment *i.e.* a network of $10,000$ nodes with 4096-byte blocks. *Figure 6.6* shows the results related to this

series of experiments. Note that the *kool* implementation does not send the block to every one of the write quorum's nodes. Instead, a single instance is sent to one storage node while the others are sent a hash of the block along with the network address of the node with the complete block. This technique drastically improves the client's performance though complicating the process on the server's side. Note that this process could further be improved to behave in a way similar to *Bittorrent* [Coh03]'s dissemination protocol.

These experiements show that both the overlay network and the distributed hash table behaves as expected *i.e.* in a scalable way. The following studies the performance of the distributed hash table when it comes to blocks composing an *Infinit* file system.

Table 6.3 provides a summary of the blocks composing the *Infinit* peer-to-peer file system. For every logical block, the table provides *(i)* its median size and *(ii)* the latency of the process consisting in retrieving this particular block from the distributed hash table. Note that next to the latency are provided, between parentheses, the *retrieving* and *validation* time. The retrieving time represents the sum of the network latencies involved in the fetching process. The validation time however is composed of the times spent verifying the integrity and authenticity of the block instances. Note that this experiment has been carried out in a $kool^2$ network populated by $10,000$ nodes with a fault-tolerance factor $\gamma = 3$. Therefore, whenever a client assembles a read quorum, at least seven instances of the mutable block are transferred back to the client while the `Validate`(α, β) routine is invoked between four and seven times.

It is also worth noting that this evaluation considers worst case scenarios. For example, the *OKB*-based *Object* logical block's verification process depends on whether the author is the owner, a lord or a vassal. For the purpose of this evaluation, a vassal is assumed to be the author since representing the most expensive case *i.e.* the *Access* block must be fetched, the vouching lord's permission must be checked and the voucher's signature must be verified. Besides, every benchmark starts with empty caches although such optimisations would greatly improve the system's performance. For instance, fetching the *Access* block accounts for nearly 64% of the verification process mentioned above. Note however that once the *Access* block has been fetched, the subsequent requests can benefit from the cache. Therefore, out of seven $\texttt{Validate}_{\frac{Object}{OKB}}(\alpha, \beta)$, a single call will actually make a network request for the *Object*'s associated *Access* block.

The reader may also notice that *TKB* physical blocks are far larger than their *OKB* counterparts. As discussed throughout *Section 4.2*, *TKB*s embed both the public key of the knights composing the table as well as the votes of the knights having authorised the last modification. These cryptographic components are responsible

for a large portion of the size of *TKB*s. Note that for the purpose of the evaluation, *TKB*s' table is composed of 5 knights—the maximum number of writers measured from the file system trace, as illustrated by *Figure 6.2*—while embedding 3 votes *i.e.* a majority.

Let us recall that the *TKB*-specific quorum algorithm differs from the one used for other mutable blocks. Indeed, while the size of the read quorum remains the same *i.e.* $2\gamma + 1$, the client selects the proper block by identifying $\gamma + 1$ identical instances. The important aspect about this quorum algorithm is that a hash is applied onto every received instance in order to detect the identical elements. Then, once detected, the $\texttt{Validate}_{\overline{TKB}}(\alpha, \beta)$ routine is invoked in order to ensure the integrity and authenticity of the selected instance. Therefore, while the integrity and authenticity of every received instance of an *OKB* is validated, a *TKB* however is validated once. This insight is made clear in *Table 6.3*. While the validation process of *OKB*-based *Group*s takes 3.91 ms, it takes only 2.23 ms for the equivalent *TKB*-based *Group*s. Noteworthy is that this ratio is not respected for the *Object* logical block because the step consisting in fetching the *Access* block covers the rest of the validation process.

Block	**Median Size**	**Latency**
CHB		
Contents	4071 bytes	$72.02\,(^{72.01}_{0.01})$ ms
Access	1045 bytes	$56.08\,(^{56.07}_{0.01})$ ms
Data	3095 bytes	$71.42\,(^{71.41}_{0.01})$ ms
Catalog	2278 bytes	$63.24\,(^{63.23}_{0.01})$ ms
Reference	56 bytes	$46.21\,(^{46.20}_{0.01})$ ms
Members	1181 bytes	$56.23\,(^{56.22}_{0.01})$ ms
PKB		
User	367 bytes	$102.63\,(^{101.68}_{1.86})$ ms
OKB		
Object	974 bytes	$179.16\,(^{119.89}_{59.27})$ ms
Group	517 bytes	$129.04\,(^{125.13}_{3.91})$ ms
TKB		
Object	3551 bytes	$340.43\,(^{279.42}_{61.01})$ ms
Group	2177 bytes	$198.42\,(^{196.19}_{2.23})$ ms

Table 6.3: An evaluation summary of the *Infinit* blocks

6.2.3 File System

Until now, the implementation has been evaluated through the *Infinit* network component only, *i.e. Hole*. This section evaluates the *Infinit* design and implementation by taking into account the whole system architecture composed of the several components presented in *Chapter 5*.

6.2.3.1 Implementation

This section focuses on implementation aspects by discussing some design choices and analysing their impact on the performance.

First, the cryptosystems must be studied in order to evaluate their impact on the system's performance. Indeed, some cryptosystems perform some operations faster than others but at the expense of larger keys for instance. The *Infinit* file system prototype makes use of the cryptosystems which are listed in *Table 6.4* along with their respective benchmarks. Note that every experiment has been performed on a randomly generated chunk of 4096 bytes of data. These algorithms have been chosen based on the study carried out by *Busca* [Bus07] which showed that these cryptosystems were the most beneficial to systems making use of mutable blocks such as *PKB*s and *OKB*s. One can notice that the generation process of the *RSA (Rivest Shamir Adleman)* asymmetric cryptosystem is several orders of magnitude more expensive than the other operations.

Cryptosystem	Operation	Duration
RSA_{1024}	Generation	96.281 ms
	Encryption	0.315 ms
	Decryption	2.728 ms
	Signature	2.854 ms
	Verification	0.233 ms
AES_{256}	Generation	0.011 ms
	Encryption	0.072 ms
	Decryption	0.069 ms
SHA_{256}	Hash	0.005 ms

Table 6.4: Performance of the *Infinit*'s cryptosystems

In addition, *Table 6.5* provides the reader with the size of the principal cryptographic components whenever embedded in blocks such as the ones described throughout

Chapter 4. These components are assumed to be issued from the cryptosystems described above *i.e.* RSA_{1024} and SHA_{256}.

Component	Size
Public Key	151 bytes
Private Key	644 bytes
Signature	128 bytes
Hash	20 bytes

Table 6.5: The size of the principal cryptographic components

Given this information, one may wonder how cryptography impacts the file system, especially regarding the use of *OKB*s. Indeed, since *Object* blocks rely on the *OKB* physical block, whenever a file, directory or link is created, a *RSA* key pair is actually generated. This design decision is expected to drastically impact the system's overall performance and must therefore be studied.

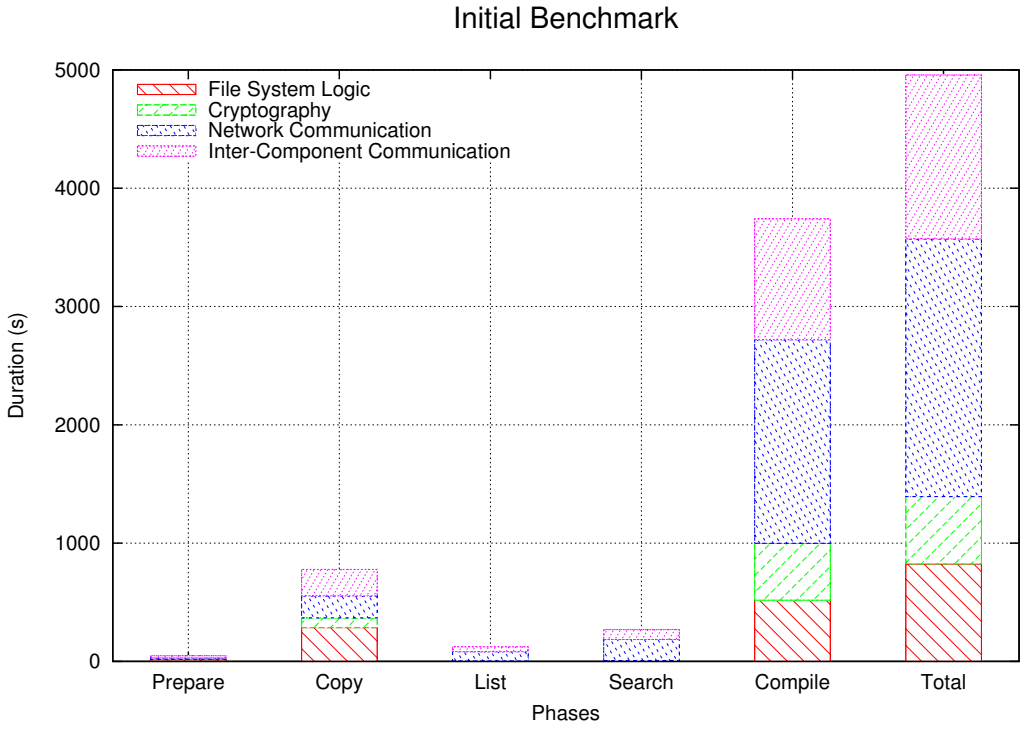


Figure 6.7: An initial benchmark with time phases

Figure 6.7 shows the result of an initial *Andrew* benchmark carried out in the *realistic* environment. For each phase, the duration is split into four sections. The *File System Logic* represents the time spent processing the file system call within one

of the *Infinit* components such as *Etoile*. The *Inter-Component Communication* represents the time spent sending messages between the *Infinit* components. The *Network Communication* and *Cryptographic* sections are self-explanatory.

Although one may have thought that cryptography would have had an enormous impact on the file system performance, it appears that inter-component communication does much more harm. The *Infinit* implementation has therefore been re-worked in order to merge all the components into a single processing unit. In addition, the *OKB* physical construct has been replaced by another physical construct referred to as *IB (Imprint Block)* which is detailed next.

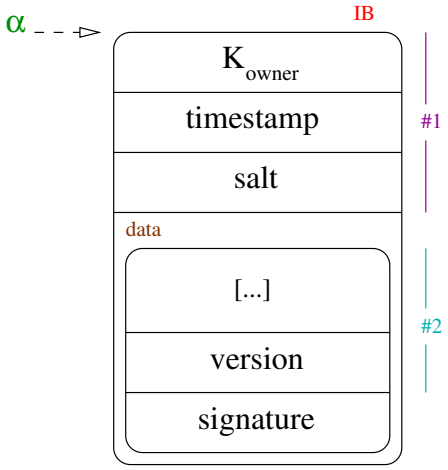


Figure 6.8: The representation of an *IB*

Figure 6.8 depicts the *IB* internal organisation. This construct ensures that, for a given user, every block created has a unique address by applying a one-way function on the tuple $(K_{owner}, timestamp, salt)$. In addition, *Algorithms 35, 36 and 37* illustrate an *IB*'s set-up, seal and validation processes, respectively.

Require: (K_{user}, k_{user}), the user's personal key pair

1. $\beta.K_{owner} \leftarrow K_{user}$
2. $\beta.timestamp \leftarrow$ **retrieve** current timestamp
3. $\beta.salt \leftarrow$ **generate** random salt
4. $\beta.data.version \leftarrow 0$
5. $\alpha \leftarrow h(\beta.\sharp 1)$
6. **return** α

Algorithm 35: $\underline{\texttt{Setup}}_{IB}(\beta) \longrightarrow \alpha$

Require: (K_{user}, k_{user}), the user's personal key pair

1. $\beta.data.signature \leftarrow h(\beta.\sharp 2)^{\widetilde{k_{user}}}$

Algorithm 36: $\texttt{Seal}_{\overline{IB}}(\alpha, \beta)$

1. **if** $\alpha \neq h(\beta.\sharp 1)$ **then**
2. **error** *"the address does not match the block"*
3. **end if**
4. **if** $\beta.data.signature^{\beta.\widetilde{K_{owner}}} \neq h(\beta.\sharp 2)$ **then**
5. **error** *"the data signature is invalid"*
6. **end if**

Algorithm 37: $\texttt{Validate}_{\overline{IB}}(\alpha, \beta)$

As the reader can notice, the *IB* construct is interesting because it removes the key pair generation but also simplifies the block verification process which now requires a single signature verification.

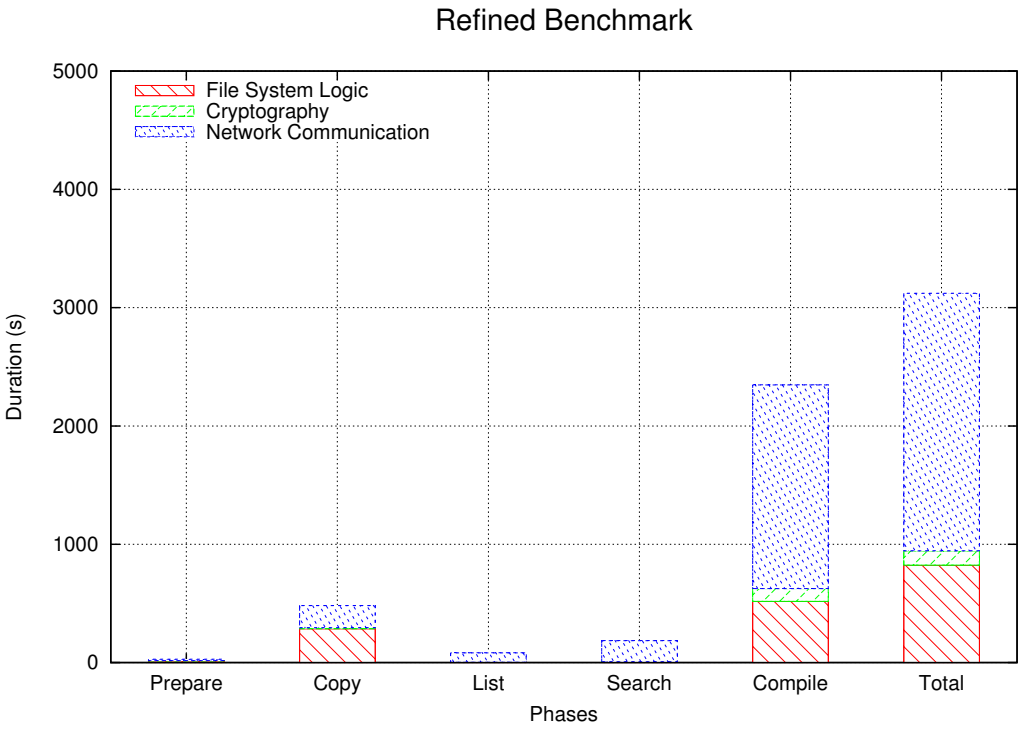


Figure 6.9: A refined benchmark with time phases

Figure 6.9 illustrates the performance gain resulting from the components merging along with the introduction of the *Imprint Block*. While the inter-components communication has completely been removed, the cryptography-specific optimisation by itself led to a 9.78% gain in overall performance.

The following evaluates the *Infinit* file system through a set of experiments which aim at comparing *Infinit*, in several configurations, with *NFS* [Osa88]. Since similar systems have also performed such a comparative evaluation, the performance of *Infinit* can be indirectly compared with similar projects, especially *Pastis* [mBPS05].

Note that for this bencharmk, a couple of caching optimisations have been activated. The first one, known as *"block cache"*, is a common technique which consists in keeping a local copy of accessed blocks for some time in order to speed up future accesses. The second one, referred to as *"path cache"*, consists in caching the relations between logical paths such as `/bin/ls` and the address of the associated *Object* block. The impact of such an optimisation is quite important since the system no longer has to fetch the intermediate directory objects in order to resolve a recently accessed path.

As shown in *Figure 6.10*, this experiment compares the performance of *NFS* against *Infinit* using three different *Hole* implementations, as detailed in *Section 5.2.6*. Note that the *kool*^{2} implementation relies on the *realistic* environment, as presented in *Section 6.1.1*, which is composed of 16 nodes located throughout the world. Also, the quorum algorithm has been configured to tolerate up to $\gamma = 3$ Byzantine nodes implying that every block is constantly replicated on 10 storage nodes.

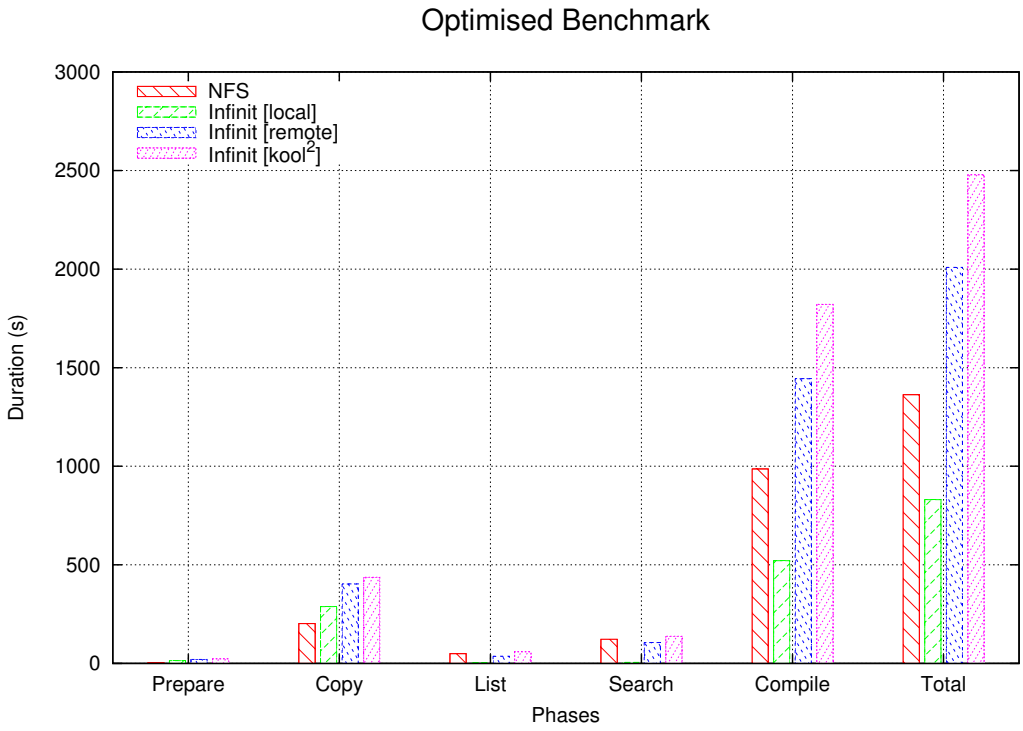


Figure 6.10: Comparison between *NFS* and several *Infinit* configurations

The results of this experiment show that *Infinit* generally performs well with execution times approximately 1.8 times slower than *NFS*. By comparison, *Ivy* [MMGC02]

is 2 to 3 times slower, depending on the configuration. Unfortunately, *Pangaea* [SKKM02], which exhibits better performance than *NFS*, cannot be considered for comparison because of only accessing local copies while not being designed to tolerate Byzantine behaviours. Likewise, *FARSITE* [ABC+02] has been compared with *NTFS (NT File System)* from which it draws its features. *Pastis* [mBPS05] however did perform a complete analysis of its performance to which *Infinit* can be compared.

Pastis performs slightly better than *Infinit*, between 1.4 and 1.9 slower than *NFS*, depending on the client's consistency model. However, several configuration parameters as well as system characteristics must be taken into account. Firstly, *Pastis* relies on *Pastry* [RD01a] which has a slightly longer routing path than *kool*^{2}'s. This characteristic however favours *Pastis* regarding the experimental results. On the other hand, *Infinit* suffers from more communication due to a higher replication factor. Indeed, every block is replicated ten times in *Infinit* against four in *Pastis*. Also, *Pastis* relies on a single mutable block for representing file system objects while providing some means of access control. *Infinit* however stores access control information in a separate immutable block, the *Access* logical block. Thus, following every *Object* block retrieval, the *Access* block must be fetched as well, hence incurring additional network overhead. Finally, while *Infinit*'s architecture has been realistically deployed through the use of *FUSE* [FUS], *Pastis*, although supporting *FUSE* as well, performed its evaluation by providing a specific *Java* [JAV] interface. These differences may account for *Pastis*' slightly better performance results.

6.2.3.2 Design

This section focuses on evaluating the *Infinit* file system in a production environment in order to analyse both the accessibility of the file system objects along with the concurrency of the updates.

Accessibility

In this first phase, the accessibility of the file system objects is analysed. More precisely, the aim of this evaluation is to study the file system access control mechanism by measuring the rate of accesses successfully performed on files, directories *etc.* that have been shared in reading, as shown in *Table 6.1*.

For that purpose, an *Infinit* file system environment is created based on the campus topology. Then, user entities are generated according to the file system trace. Unfortunately, the group entities could not be generated that easily since, as for most centralised file systems, the trace's groups are managed following the *MAC* access control policy, *i.e.* users are not granted the right to create or even manage groups in the system.

Since *Infinit* complies to the opposite policy, *i.e.* *DAC*, the groups had to be generated through another technique. For every file system object that has been shared in reading, a group is generated that includes all such users with read permission. Note that such a group generation process implies that, in the *Infinit* environment, every such file, directory *etc.* actually references groups only. In other words, such file system objects mainly reference vassals with only the object owner acting as lord. Although this decision represents the worst case scenario, given the *Infinit* design presented in *Chapter 4*, the system could not have decided automatically how many lords to create, or even which users to include as a lord.

Noteworthy is that there is an exception when it comes to file system objects that are shared in reading with a single user other than the owner, in which case no group is created, *i.e.* the single user granted permission acts as a lord.

In order to alleviate this extreme arrangement in which a group is created for every set of readers, several optimisations have been activated such as the proactive distribution of the key used for encrypting the *Contents* block. In addition, users acting as vassals can request the key from other vassals, should one of them be connected at that time.

Figure 6.11 summarises the 156,729 file system object accesses according to the entity requesting it. This figure shows that most accesses are actually performed by the owner along with lords. The large number of accesses performed by lords can probably be explained by the fact that 99.8% of the 267,955,200 shared file system objects are shared with a single user *i.e.* a lord.

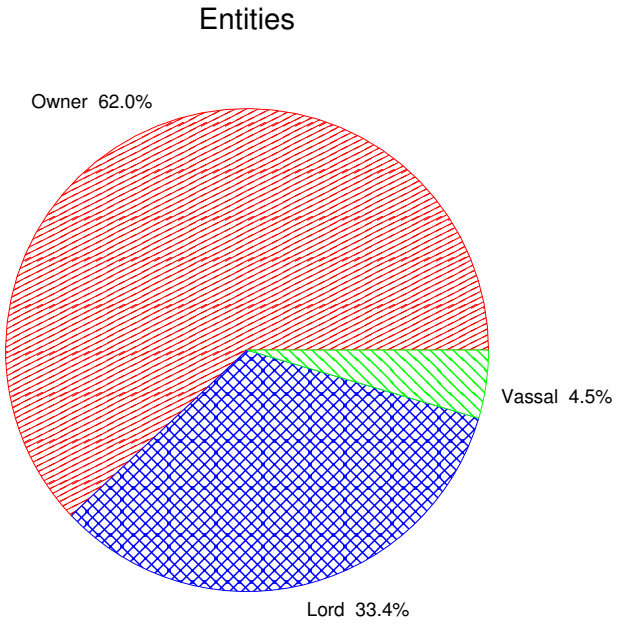


Figure 6.11: Summary of the accesses according to the entity

Finally, *Figure 6.12* focuses on the vassals' accesses by analysing the file system's state when such accesses are requested. As shown on the figure, it appears that 43.4% of the the accesses made by vassals could be performed because the vassal already had received the key or could contact the owner to retrieve it. This high number can probably be explained by social interactions where students, for example, actually seated next to each other, decide to share a file. Such scenarios might explain the availability of both parties during the object creation or the access. The figure also demonstrates the efficiency of the proactive distribution optimisation which enables 54.5% of the accesses to be performed by retrieving the key from another vassal which happened to be logged in when the object was created. Unfortunately however, 2% of the accesses could not be performed because of the unavailability of the owner and the impossibility to retrieve the key from another vassal.

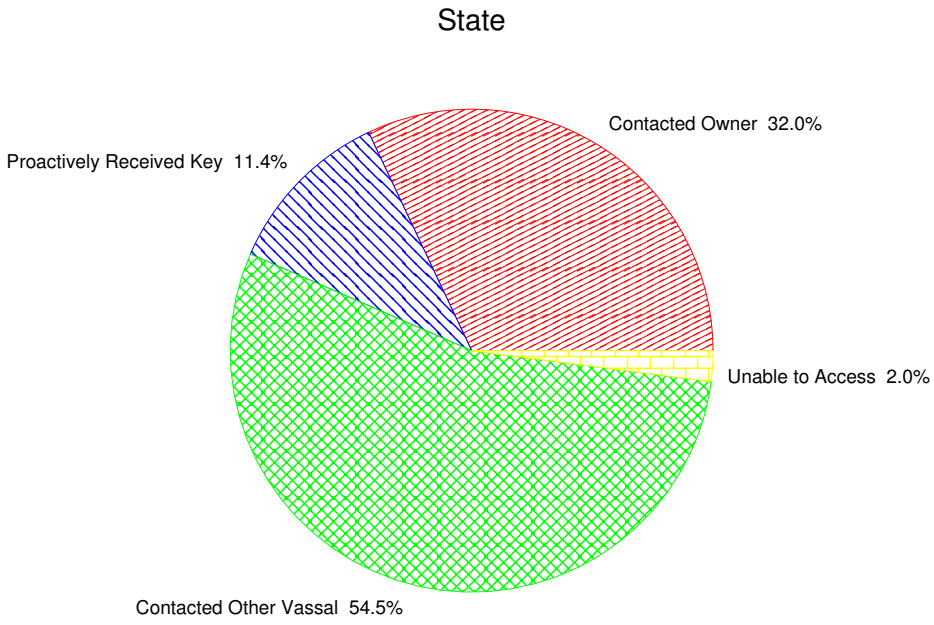


Figure 6.12: State of the accesses performed by the vassals

Concurrency

The second phase aims at analysing the update conflicts especially when it comes to the administration scheme described in *Chapter 4*.

Let us recall that the administration mechanism requires file system objects to be associated with a set of users responsible for taking management decisions including the modification of the object's content. As discussed in *Section 4.2*, although this design is extremely flexible, these special users, referred to as knights, may not be well-connected enough in order to handle the flow of requests generated by the object's lords and vassals.

For the purpose of this evaluation, an *Infinit* file system environment is created for which some file system objects will rely on the administration mechanism. Indeed, for every object being writable by multiple users, a *TKB*-based *Object* is created while all the writers are included in the table of knights. Note that such an organisation represents the worst-case scenario. Indeed, in a deployed *Infinit* environment, users would likely elect another user to the grade of knight assuming this user is well-connected and involved in the system. This condition is crucial for the administration requests to be processed as quickly as possible.

The following benchmark analyses every one of the 632 modification requests made to file system objects which have been shared with other writers, as summarised in *Table 6.2*.

Figure 6.13 summarises the results by considering several time frames. A time frame indicates how much time a knight needs to be connected in order to process a request *i.e.* issue a vote. For instance, the shortest time frame considered is 1 minute, which indicates that a user acting as a knight will take, on average, 1 minute to issue her vote. Such time frames emulate the fact that users may take some time to *(i)* notice the fact that a request has been made and *(ii)* consider the request and vote. The figure shows that as the time frame increases, consensus take less time to be reached.

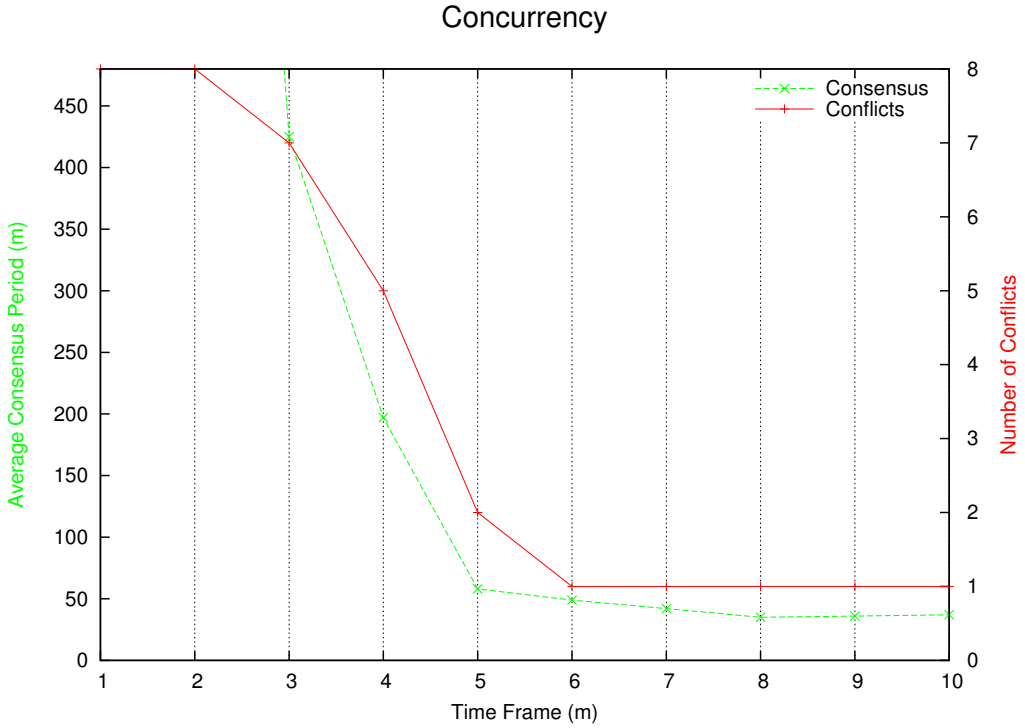


Figure 6.13: Concurrency analysis depending on the knights' reactivity

In addition, the figure depicts the number of conflicts detected. A conflict can be identified whenever an object's knights have not yet reached consensus on a request

while a new request is issued. The reader can notice that the number of such conflicts is quite low: 8. It is fair to assume that users relying on the file system's access control mechanism do not intend to update files in a concurrent manner. For that purpose, users tend to rely on specific tools such as *Subversion*, *Git etc.* as further analyses confirm.

The number of such conflicts appears to decrease as consensus is reached more quickly. Note however that a conflict remains anyway. A detailed analysis showed that this conflict comes from users updating, a few seconds apart, the same directory by creating different files. Unfortunately, the knights do not have the time to vote and reach consensus within such an extremely short period of time. This conflict illustrates the core drawback of the given administration scheme.

⋆ ⋆
⋆

To conclude, although *Infinit* exhibits decent performance results, one must be reminded that many components have been implemented in their simplest form possible. In addition, a number of optimisations could be applied in order to decrease the impact of some of the design's characteristics. Beside optimisations that could be applied both from the design and implementation perspectives, interesting concepts could also be borrowed from similar systems. For example, *Plutus* [KRS+03] introduced the idea of *filegroup*s so that objects sharing similar access control rules could be optimised in several ways: *(i)* fewer keys are generated since shared among the objects of the same *filegroup* and, in turn, *(ii)* less space is used for metadata while the system can rely further on caching techniques. Therefore, although *Infinit* performs generally well, many additional design and implementation improvements could be made in order to bring its performance closer to those of *NFS* [Osa88].

The analysis of a file system trace proved that the *Infinit* design presented in *Chapter 4* could actually work in a production environment. The access control mechanism proved itself efficient given extreme configurations including the worst-case scenario in which object owners grant access to users solely through the use of group entities. The benchmark illustrated that the extremely small number of lords along with their unpredictable connectivity could be overcome by applying several optimisations such as the proactive distribution of the key used for encrypting the object's content. Although the analysis revealed that some accesses could not be performed due to the unavailability of users from which to retrieve the key, it is fair to assume that an *Infinit* production environment would behave differently. For instance, by

relying on an open peer-to-peer network, users would be more likely to be connected through one of their computing devices which in turn would alleviate the discussed accessibility issues.

Although the access control mechanism deployment was proven successful, the administration scheme suffered from the knights' reactivity to treat the incoming requests. This trait became critical when considering concurrent updates since, as described in *Section 4.2*, administrative requests cannot be treated in parallel. Should such a conflict occur, the user would have to delay her request and re-submit it later, which obviously could represent an important hurdle for users. Note however that the evaluation considered the worst-case scenario in which every object being writable by multiple users was considered a cooperative object. Indeed, in practice, it is very likely that the number of such objects would remain extremely small, the administration scheme being used for critical objects only.

Chapter 7

Conclusion

The peer-to-peer model has shown itself to be a powerful paradigm for the design of large-scale, adaptative and highly-resilient systems. Over the last decades, many peer-to-peer systems have been conceived offering services as diverse as telephone, video streaming or even file sharing through the popular *Bittorrent* [Coh03] application. On the other hand, the *file* abstraction provided by the hierarchical file system interface has become the common way for organising, naming and accessing digital data. As mentioned in *Section 3.2*, these paradigms can be combined in order to build a file system benefiting from the properties inherent to the peer-to-peer model such as scalability, fault-tolerance, durability, availability and so forth.

Throughout this thesis, the author has shown it possible to design a modern storage system based on these paradigms in order to ensure a certain number of fundamental properties, defined in *Section 3.1*. Therefore, an access control and an administration scheme have been designed, paving the way for the implementation of such a system.

Noteworthy is that this work has been based on several assumptions which could very well be challenged by other authors. Note however that such decisions have been justified in *Section 3.4*.

The first assumption relates to the peer-to-peer environment which implies a number of properties such as the untrustworthiness of the computers populating the network, the decentralised and symmetric behaviour of those nodes as well as the required scalability of the underlying network protocols such as locating a node responsible for a given identifier. Besides, the network's untrustworthiness and symmetry implies that the data blocks associated with an identifier must be self-certified so that anyone can distinguish a valid from an illegitimately forged block. Note however that the administration scheme presented in *Section 4.2* introduced blocks, referred to as *TKB*s, which do not conform to these principles, thus violating both the symmetry and self-certification properties.

The nodes and especially their connectivity to the peer-to-peer network constitutes

another assumption. *Section 4.1.2* showed that the fundamental properties were too constraining for an access control scheme to be designed in such an environment. Therefore, the connectivity requirement was loosened as thought to be the more flexible parameter. Most research interested in such aspects of peer-to-peer networks has focused on the nodes' churn ratio. The access control scheme presented in this document relies on the connectivity of users which, in such a modern system, can be connected to the system through multiple computing devices being computers, mobile phones, tablets, netbooks *etc.*

The third and final assumption stipulates that the distributed hash table upon which the presented file system relies should ensure consistency among the replicas through a quorum-based protocol. Although similar projects such as *CFS* [DKK+01] or *Pastis* [mBPS05] have also made this assumption, most Byzantine-fault-tolerant distributed systems tend to make use of agreement algorithms such as *BFT* [CL99] or *Paxos* [Lam98] because these algorithms provide far more flexibility than their quorum counterparts. On the other hand, quorum algorithms are well suited for distributed file systems because they rely on the basic operations consisting of reading and writing data items. As such, file systems do not require the underlying storage layer to provide advanced functionalities. All in all, quorum algorithms imply self-certification which in turn requires more cryptographic operations while agreement protocols require storage nodes to exchange more messages in order to achieve consensus. However, and as shown in *Chapter 6*, the cryptographic operations account for an extremely small portion of the retrieval and storing processes, hence confirming the initial assumption regarding the better performance of quorum algorithms.

The contributions of this work are threefold. First, the functionalities such a modern storage system should provide to end-users have been defined though some have intentionally been left for future work. In addition, the system's properties such as untrustworthiness, decentralisation, symmetry, self-certification *etc.* have been inferred from the peer-to-peer file system's paradigms. Second, the fundamental file system components such as file, directory, user, group *etc.* have been defined through the design of an access control and administration scheme. Unlike previous projects such as *Chefs* [Fu05], *Plutus* [KRS+03] and *Pastis* [mBPS05], the access control scheme has been designed for large-scale decentralised and untrustworthy environments while providing users with the means to express access control rules in a very flexible way. In addition, the administration scheme allows users to request administrative tasks while preventing a single user from taking complete control over the system. Third, a prototype implementation has been developed proving feasible the deployment of such a system. This prototype has been developed so as to provide a modular architecture enabling the user to set up the system according to its device's hardware characteristics as well as the user's preferences.

The evaluation carried out in *Chapter 6* shows that *Infinit* performs generally well, especially compared to similar systems such as *CFS* [DKK+01], *Ivy* [MMGC02] and *Pastis* [mBPS05]. Interestingly, the access control mechanism has been proven efficient and robust especially in extreme environments with low connectivity. However, a long-term analysis would still have to be performed in a large-scale realistic environment in order to validate the qualitative aspects of the system especially regarding the administration scheme which suffers from the knights' reactivity when it comes to concurrent updates.

Finally, although this thesis provides the fundamental components for the implementation of a large-scale decentralised and Byzantine-fault-tolerant storage system, some properties have been left as future work, such as anonymity and versioning, while other aspects have been voluntarily ignored including garbage collection, consistency models, advanced quorum algorithms, concurrency conflicts resolution and many more. Although every one of these topics has been tackled through other research projects, *Infinit* could not be considered complete without taking such design factors into account.

List of Figures

1.1 A worldwide storage infrastructure 4

2.1 A flat unstructured overlay network 9
2.2 A two-level hybrid overlay network 10
2.3 A ring-based structured overlay network 12
2.4 A *Chord* network of degree 5 with 17 nodes 15
2.5 A *Kelips* network for 36 nodes 17
2.6 A small-world-based social overlay network 19
2.7 The replication-based *DHash* distributed hash table 22
2.8 The *Paxos* agreement protocol 23
2.9 Three *Gifford* quorum configurations 24
2.10 *Pangaea* file system representation 27
2.11 *OceanStore*'s organisation 29
2.12 The *FARSITE* architecture 30
2.13 The *CFS* hierarchical organisation 32
2.14 The *Ivy* log-based representation 34
2.15 The *Plutus*' keys, locks and groups 36
2.16 The *Pastis* organisation 37

3.1 A three-step representation of a symmetric quorum-based system . . 50
3.2 A three-step representation of an asymmetric quorum-based system . 51

4.1 The representation of a *CHB* 63
4.2 The representation of a *PKB* 64
4.3 The representation of an *OKB* 65

4.4 The representation of a *PKB*-based *User* block 67
4.5 The representation of an *OKB*-based *Group* block 69
4.6 The representation of an *OKB*-based *Object* block 72
4.7 A graph showing the relations between P_O and ρ 82
4.8 The representation of a *TKB* 94
4.9 A scenario illustrating the *TKB*-specific quorum algorithm 101
4.10 The representation of a *TKB*-based *Object* block 103
4.11 The representation of an *TKB*-based *Group* block 106

5.1 A file representation 121
5.2 A directory representation 122
5.3 A link representation 122
5.4 The *Infinit* system-wide hierarchical representation 124
5.5 The architecture of an *Infinit* node 125
5.6 The internals of the *Etoile* component 135
5.7 An example of a $kool^3$ network 143

6.1 General information regarding the users' sharing behaviours 151
6.2 General information regarding the user's cooperative behaviours . . . 151
6.3 The performance of the overlay network's `Lookup`(ι) routine 153
6.4 The performance of the immutable-specific `Get`(α) routine 154
6.5 The performance of the mutable-specific `Gather`(α) routine 155
6.6 The performance of the `Put`(α, β) routine 155
6.7 An initial benchmark with time phases 159
6.8 The representation of an *IB* 160
6.9 A refined benchmark with time phases 161
6.10 Comparison between *NFS* and several *Infinit* configurations 162
6.11 Summary of the accesses according to the entity 164
6.12 State of the accesses performed by the vassals 165
6.13 Concurrency analysis depending on the knights' reactivity 166

List of Tables

4.1 A summary of the permissions in the file system 95

5.1 *kool* parameters 142
5.2 *kool* formulas 143
5.3 Comparison of the *kool* configurations 144

6.1 General information regarding the users and files 149
6.2 General information regarding the nodes and users connectivity . . . 150
6.3 An evaluation summary of the *Infinit* blocks 157
6.4 Performance of the *Infinit*'s cryptosystems 158
6.5 The size of the principal cryptographic components 159

Listings

5.1 An example of hierarchical namespace . . . 123

5.2 The `PublicKey::Serialize()` method . . . 127

5.3 The `KeyPair::Extract()` method . . . 127

5.4 The *Base64* `Unique` representation of a block address . . . 128

5.5 An illustration of fibers . . . 129

5.6 The message definition process . . . 131

5.7 The message registration process . . . 131

5.8 The *PIG*'s `rmdir()` *POSIX* system call . . . 132

5.9 The `Agent::Decrypt()` method . . . 134

5.10 The `Agent::Sign()` method . . . 134

5.11 *Etoile*'s *wall* message definitions for directory objects . . . 136

5.12 *Etoile*'s *wall* handler definitions for directory objects . . . 136

5.13 The *components* unit's `Directory::Remove()` method . . . 138

5.14 The `Object` class . . . 139

5.15 The `Hole` component's interface . . . 140

6.1 The *Andrew* benchmark . . . 147

List of Algorithms

1 $\texttt{Setup}_{\overline{CHB}}(\beta) \longrightarrow \alpha$ . . . 63
2 $\texttt{Seal}_{\overline{CHB}}(\alpha, \beta)$ . . . 63
3 $\texttt{Validate}_{\overline{CHB}}(\alpha, \beta)$ . . . 63
4 $\texttt{Setup}_{\overline{PKB}}(\beta) \longrightarrow \alpha$ . . . 64
5 $\texttt{Seal}_{\overline{PKB}}(\alpha, \beta)$ . . . 65
6 $\texttt{Validate}_{\overline{PKB}}(\alpha, \beta)$ . . . 65
7 $\texttt{Setup}_{\overline{OKB}}(\beta) \longrightarrow \alpha$ . . . 66
8 $\texttt{Seal}_{\overline{OKB}}(\alpha, \beta)$ . . . 66
9 $\texttt{Validate}_{\overline{OKB}}(\alpha, \beta)$ . . . 66
10 $\texttt{Setup}_{\frac{User}{PKB}}(\beta) \longrightarrow \alpha$ . . . 68
11 $\texttt{Seal}_{\frac{User}{PKB}}(\alpha, \beta)$ . . . 68
12 $\texttt{Validate}_{\frac{User}{PKB}}(\alpha, \beta)$ . . . 68
13 $\texttt{Setup}_{\frac{Object}{OKB}}(\beta) \longrightarrow \alpha$ . . . 73
14 $\texttt{Seal}_{\frac{Object[data]}{OKB}}(\alpha, \beta)$ . . . 73
15 $\texttt{Seal}_{\frac{Object[meta]}{OKB}}(\alpha, \beta)$ . . . 73
16 $\texttt{Validate}_{\frac{Object}{OKB}}(\alpha, \beta)$ . . . 74
17 $\texttt{Govern}(\alpha, \psi)$ . . . 76
18 $\texttt{Read}(\alpha, \lambda) \longrightarrow \delta$ . . . 77
19 $\texttt{Write}(\alpha, \lambda, \delta)$ . . . 79
20 $\texttt{Setup}_{\overline{TKB}}(\beta) \longrightarrow \alpha$ . . . 95
21 $\texttt{Seal}_{\overline{TKB}}(\alpha, \beta)$ . . . 95
22 $\texttt{Validate}[client]_{\overline{TKB}}(\alpha, \beta)$ — client side . . . 96
23 $\texttt{Validate}[server]_{\overline{TKB}}(\alpha, \beta)$ — server side . . . 100
24 $\texttt{Setup}_{\frac{Object}{TKB}}(\beta) \longrightarrow \alpha$ . . . 104
25 $\texttt{Seal}_{\frac{Object[data]}{TKB}}(\alpha, \beta)$ . . . 104
26 $\texttt{Seal}_{\frac{Object[meta]}{TKB}}(\alpha, \beta)$ . . . 104
27 $\texttt{Validate}[client]_{\frac{Object}{TKB}}(\alpha, \beta)$ — client side . . . 105
28 $\texttt{Elect}(\alpha, \theta)$ . . . 110
29 $\texttt{Govern}(\alpha, \psi)$ . . . 111
30 $\texttt{Read}(\alpha, \lambda) \longrightarrow \delta$ . . . 112
31 $\texttt{Write}(\alpha, \lambda, \delta)$ . . . 113
32 $\texttt{Manage}(\alpha, \theta)$ . . . 114

33 $\texttt{Edit}(\alpha, \psi)$ 115
34 $\texttt{Transfer}(\alpha, \mu)$ 115
35 $\texttt{Setup}_{\overline{IB}}(\beta) \longrightarrow \alpha$ 160
36 $\texttt{Seal}_{\overline{IB}}(\alpha, \beta)$ 161
37 $\texttt{Validate}_{\overline{IB}}(\alpha, \beta)$ 161

Bibliography

[ABC+02] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *In Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–14, 2002.

[ACMR02] Sameer Ajmani, Dwaine E. Clarke, Chuang-Hue Moh, and Steven Richman. ConChord: Cooperative SDSI certificate storage and name resolution. In *In First International Workshop on Peer-to-Peer Systems*, pages 141–154, 2002.

[ADN+95] Thomas Anderson, Michael Dahlin, Jeanna Neefe, David P. Terson, Drew Roselli, and Randolph Wang. Serverless network file systems. In *In Proceedings of the 15th Symposium on Operating System Principles. ACM*, pages 109–126, Copper Mountain Resort, Colorado, December 1995.

[AHT+02] A. Adya, J. Howell, M. Theimer, W. Bolosky, and J. Douceur. Cooperative task management without manual stack management, 2002.

[AJ92] Gagan Agrawal and Pankaj Jalote. An efficient protocol for voting in distributed systems. In *International Conference on Distributed Computing Systems*, pages 640–647, 1992.

[AMN01] Michel Abdalla, Sara K. Miner, and Chanathip Namprempre. Forward-secure threshold signature schemes. In *Proceedings of the 2001 Conference on Topics in Cryptology: The Cryptographer's Track at RSA*, CT-RSA 2001, pages 441–456, London, UK, UK, 2001. Springer-Verlag.

[And04] Ken H. And. Small world overlay P2P networks, 2004.

[App] http://www.me.com.

[BBB+04] Jean-Michel Busca, Marin Bertier, Fatima Belkouch, Pierre Sens, and Luciana Arantes. A performance evaluation of a quorum-based state-machine replication algorithm for computing grids. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing*, pages 116–123, Washington, DC, USA, 2004. IEEE Computer Society.

[BCK+09] Gal Badishi, Germano Caronni, Idit Keidar, Raphael Rom, and Glenn Scott. Deleting files in the celeste peer-to-peer storage system. *Journal of Parallel and Distributed Computing*, 69(7):613–622, July 2009.

[BDET00] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. *SIGMETRICS Perform. Eval. Rev.*, 28(1):34–43, June 2000.

[BF08] João Barreto and Paulo Ferreira. The obscure nature of epidemic quorum systems. In *Proceedings of the 9th workshop on Mobile computing systems and applications*, HotMobile '08, pages 69–73, New York, NY, USA, 2008. ACM.

[BJZH04] Ali R. Butt, Troy A. Johnson, Yili Zheng, and Charlie Y. Hu. Kosha: A Peer-to-Peer enhancement for the network file system. In *The International Conference for High Performance Computing and Communications (SC2004)*, page 51, November 2004.

[BLV05] A. Blanc, Yi-Kai Liu, and A. Vahdat. Designing incentives for peer-to-peer routing. volume 1, pages 374–385 vol. 1, March 2005.

[Box] http://www.box.net.

[BS10] Fatemeh Borran and André Schiper. A leader-free byzantine consensus algorithm. In *Proceedings of the 11th international conference on Distributed computing and networking*, ICDCN'10, pages 67–78, Berlin, Heidelberg, 2010. Springer-Verlag.

[BTC+04] Ranjita Bhagwan, Kiran Tati, Yu C. Cheng, Stefan Savage, and Geoffrey M. Voelker. Total recall: system support for automated availability management. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, page 25, Berkeley, CA, USA, 2004. USENIX Association.

[Bus07] Jean-Michel Busca. *Pastis : Un Système Pair à Pair de Gestion de Fichiers*. PhD thesis, Université Pierre et Marie Curie, 2007.

[CCB07] James Cipar, Mark D. Corner, and Emery D. Berger. TFS: a transparent file system for contributory storage. In *FAST'07: Proceedings of the 5th conference on USENIX Conference on File and Storage Technologies*, page 28, Berkeley, CA, USA, 2007. USENIX Association.

[CCR05] M. Castro, M. Costa, and A. Rowstron. Debunking some myths about structured and unstructured overlays. In *2nd Symposium on Networked Systems Design and Implementation (NSDI'05)*, pages 85–98, 2005.

[CDG+02] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Secure routing for structured peer-to-peer overlay networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):299–314, 2002.

[CFK03] Edith Cohen, Amos Fiat, and Haim Kaplan. A case for associative peer to peer overlays. *SIGCOMM Comput. Commun. Rev.*, 33:95–100, January 2003.

[CGKV08] Gregory Chockler, Rachid Guerraoui, Idit Keidar, and Marko Vukolic. Reliable distributed storage. *IEEE Computer*, 2008.

[CGM02] Brian F. Cooper and Hector Garcia-Molina. Bidding for storage space in a peer-to-peer data preservation system. In *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, ICDCS '02, pages 372–, Washington, DC, USA, 2002. IEEE Computer Society.

[CL99] Castro and Liskov. Practical byzantine fault tolerance. In *OSDI: Symposium on Operating Systems Design and Implementation*, pages 173–186. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999.

[Cla83] Anne-Marie G. Claybrook. Directions in computer security. In *Proceedings of the 1983 annual conference on Computers : Extending the human resource*, ACM '83, pages 42–, New York, NY, USA, 1983. ACM.

[CLL07] Chin-Chen Chang, Chih-Yang Lin, and Keng-Chu Lin. Simple efficient mutual anonymity protocols for peer-to-peer network based on primitive roots. *J. Netw. Comput. Appl.*, 30:662–676, April 2007.

[CN03] L. Cox and B. Noble. Samsara: Honor among thieves in Peer-to-Peer storage. *ACM SIGOPS Operating Systems Review*, 37(5):120–132, 2003.

[Coh03] Bram Cohen. Incentives build robustness in BitTorrent, 2003.

[Com85] Douglas E. Comer. Domain names (panel session, abstract only): hierarchy in need of organization. *SIGCOMM Comput. Commun. Rev.*, 15:72–, September 1985.

[CRB+03] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making gnutella-like p2p systems scalable. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '03, pages 407–418, New York, NY, USA, 2003. ACM.

[CRS05] Germano Caronni, Raphael Rom, and Glenn Scott. Maintaining object ordering in a shared p2p storage environment. In *Proceedings of the Third IEEE International Security in Storage Workshop*, pages 52–62, Washington, DC, USA, 2005. IEEE Computer Society.

[CSP07] Lásaro Jonas Camargos, Rodrigo Malta Schmidt, and Fernando Pedone. Multicoordinated paxos. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 316–317, New York, NY, USA, 2007. ACM.

[CSWH01] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: a distributed anonymous information storage and retrieval system. In *International workshop on Designing privacy enhancing technologies*, pages 46–66. Springer-Verlag New York, Inc., 2001.

[DAB+02] John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 617–624, Washington, DC, USA, 2002. IEEE Computer Society.

[dALF10] Francisco de Asis Lopez-Fuentes. A routing scheme for content localization in peer-to-peer networks. In *Proceedings of the 2010 IEEE Electronics, Robotics and Automotive Mechanics Conference*, CERMA '10, pages 249–254, Washington, DC, USA, 2010. IEEE Computer Society.

[DB99] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *SIGMETRICS '99: Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 59–70, New York, NY, USA, 1999. ACM.

[DFM01] Roger Dingledine, Michael J. Freedman, and David Molnar. The free haven project: distributed anonymous storage service. In *International*

workshop on Designing privacy enhancing technologies, pages 67–95, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[DGWR07] A. G. Dimakis, P. B. Godfrey, M. J. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 2000–2008, May 2007.

[DHA03] Anwitaman Datta, Manfred Hauswirth, and Karl Aberer. Updates in highly unreliable, replicated Peer-to-Peer systems. In *23rd International Conference on Distributed Computing Systems (ICDCS 2003)*, pages 76–87, Providence, Rhode Island, USA, May 2003.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41:205–220, October 2007.

[DKK+01] Frank Dabek, Frans M. Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[DLLKA05] George Danezis, Chris Lesniewski-Laas, Frans M. Kaashoek, and Ross Anderson. Sybil-Resistant DHT routing. pages 305–318. 2005.

[DMM08] John Day, Ibrahim Matta, and Karim Mattar. Networking is ipc: a guiding principle to a better internet. In *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, pages 67:1–67:6, New York, NY, USA, 2008. ACM.

[DMS03] Roger Dingledine, Nick Mathewson, and Paul Syverson. Reputation in P2P anonymity systems, 2003.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, pages 303–320, San Diego, CA, USA, August 2004.

[Dou02] John R. Douceur. The sybil attack. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 251–260, London, UK, 2002. Springer-Verlag.

[Dro] http://www.getdropbox.com.

[DW01] John R. Douceur and Roger P. Wattenhofer. Optimizing file availability in a secure serverless distributed file system. 2001.

[DZD+03] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and Stoica. Towards a common API for structured Peer-to-Peer overlays. In *International Workshop on Peer-to-Peer Systems*, 2003.

[FJG06] Ronaldo A. Ferreira, Suresh Jagannathan, and Ananth Grama. Locality in structured peer-to-peer networks. *J. Parallel Distrib. Comput.*, 66:257–273, February 2006.

[FKK06] Kevin Fu, Seny Kamaram, and Yoshi Kohno. Key regression: Enabling efficient key distribution for secure distributed storage. In *Network and Distributed System Security Symposium (NDSS '06)*, 2006.

[FKM02] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. *ACM Trans. Comput. Syst.*, 20(1):1–24, February 2002.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32:374–382, April 1985.

[Fu99] Kevin Fu. Group sharing and random access in cryptographic storage file systems. Master's thesis, Massachusetts Institute of Technology, May 1999.

[Fu05] Kevin E. Fu. *Integrity and access control in untrusted content distribution networks*. PhD thesis, Cambridge, MA, USA, 2005. AAI0808990.

[FUS] http://fuse.sf.net.

[GBL+03] Indranil Gupta, Ken Birman, Prakash Linga, Al Demers, and Robbert van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *IPTPS '03: 2nd International Workshop on Peer-to-Peer Systems*, 2003.

[GDZ+05] Lei Gao, Mike Dahlin, Jiandan Zheng, Lorenzo Alvisi, and Arun Iyengar. Dual-quorum replication for edge services. In *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, Middleware '05, pages 184–204, New York, NY, USA, 2005. Springer-Verlag New York, Inc.

[GEvS07] P. Garbacki, D. H. J. Epema, and M. van Steen. Optimizing peer relationships in a Super-Peer network. In *ICDCS '07: Proceedings of*

the 27th International Conference on Distributed Computing Systems, page 31, Washington, DC, USA, July 2007. IEEE Computer Society.

[Gif79] David K. Gifford. Weighted voting for replicated data, 1979.

[GKLQ07] Rachid Guerraoui, Dejan Kostic, Ron R. Levy, and Vivien Quema. A high throughput atomic storage algorithm. In *ICDCS '07: Proceedings of the 27th International Conference on Distributed Computing Systems*, page 19, Washington, DC, USA, 2007. IEEE Computer Society.

[HAF10] Yaser Houri, Bernhard Amann, and Thomas Fuhrmann. A quantitative analysis of redundancy schemes for peer-to- peer storage systems. In *Proceedings of the 12th international conference on Stabilization, safety, and security of distributed systems*, SSS'10, pages 519–530, Berlin, Heidelberg, 2010. Springer-Verlag.

[HAY+05] Ragib Hasan, Zahid Anwar, William Yurcik, Larry Brumbaugh, and Roy Campbell. A survey of Peer-to-Peer storage techniques for distributed file systems. In *ITCC '05: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II*, pages 205–213, Washington, DC, USA, 2005. IEEE Computer Society.

[HB11] Cyrus Harvesf and Douglas M. Blough. Replica placement for route diversity in tree-based routing distributed hash tables. *IEEE Trans. Dependable Secur. Comput.*, 8:419–433, May 2011.

[HCW10] Guowei Huang, Jiangang Chen, and Lian Wei. Routeguard: A trust-based scheme for guarding routing in structured peer-to-peer overlays. In *Proceedings of the 2010 International Conference on Communications and Mobile Computing - Volume 01*, CMC '10, pages 330–334, Washington, DC, USA, 2010. IEEE Computer Society.

[HKD07] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: practical accountability for distributed systems. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 175–188, New York, NY, USA, 2007. ACM.

[HKLF+06] S. B. Handurukande, A. M. Kermarrec, F. Le Fessant, L. Massoulié, and S. Patarin. Peer sharing behaviour in the eDonkey network, and implications for the design of server-less file sharing systems. In *EuroSys '06: Proceedings of the 2006 EuroSys conference*, pages 359–371, New York, NY, USA, 2006. ACM Press.

[HKM+88a] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, February 1988.

[HKM+88b] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, February 1988.

[HMD05] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *IN PROC. OF NSDI*, 2005.

[HP94] John S. Heidemann and Gerald J. Popek. File-system development with stackable layers. *ACM Trans. Comput. Syst.*, 12(1):58–89, 1994.

[iNe] http://topology.eecs.umich.edu/inet/.

[IRF04] Adriana Iamnitchi, Matei Ripeanu, and Ian Foster. Small-World File-Sharing communities. In *The 23rd Conference of the IEEE Communications Society (InfoCom 2004)*, Hong Kong, # 2004.

[JAV] http://www.java.com.

[JB94] Marjan Jurečič and Herbert Bunz. Exchange of patient records-prototype implementation of a security attributes service in x.500. In *Proceedings of the 2nd ACM Conference on Computer and communications security*, CCS '94, pages 30–38, New York, NY, USA, 1994. ACM.

[JGH+98] Jr, R. Guy, J. Heidemann, D. Ratner, P. Reiher, A. Goel, G. Kuenning, and G. Popek. Perspectives on optimistically replicated, Peer-to-Peer filing. *Software Practice and Experience*, February 1998.

[JXY07] Yi Jiang, Guangtao Xue, and Jinyuan You. Distributed hash table based peer-to-peer version control system for collaboration. In *Proceedings of the 10th international conference on Computer supported cooperative work in design III*, CSCWD'06, pages 489–498, Berlin, Heidelberg, 2007. Springer-Verlag.

[Kar86] P A Karger. Authentication and discretionary access control in computer networks. *Comput. Secur.*, 5:314–324, December 1986.

[KBC+00] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. In *In Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS*, 2000.

[KLW11] Andreas Klappenecker, Hyunyoung Lee, and Jennifer L. Welch. Quorum-based dynamic regular registers in systems with churn. In *Proceedings of the 3rd International Workshop on Theoretical Aspects of Dynamic Distributed Systems*, TADDS '11, pages 3–7, New York, NY, USA, 2011. ACM.

[KRS+03] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 29–42, Berkeley, CA, USA, 2003. USENIX Association.

[KS10] Jinu Kurian and Kamil Sarac. A survey on the design, applications, and enhancements of application-layer overlay networks. *ACM Comput. Surv.*, 43:5:1–5:34, December 2010.

[KSMK03] Michael Kaminsky, George Savvides, David Mazieres, and Frans M. Kaashoek. Decentralized user authentication in a global file system. *SIGOPS Oper. Syst. Rev.*, 37(5):60–73, December 2003.

[KWR06] P. Knezevic, A. Wombacher, and T. Risse. Highly Available DHTs: Keeping Data Consistency After Updates. In *4th International Workshop, AP2PC 2005, July 25, 2005, Revised Papers, Utrecht, Netherlands*, July 2006.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[Lam01] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[LCG07] Haifeng Liu, Xianglan Chen, and Yuchang Gong. Babyos: a fresh start. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education*, SIGCSE '07, pages 566–570, New York, NY, USA, 2007. ACM.

[LKMS04] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, page 9, Berkeley, CA, USA, 2004. USENIX Association.

[LMZ09] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 312–313, New York, NY, USA, 2009. ACM.

[LSG+04] Jinyang Li, Jeremy Stribling, Thomer M. Gil, Robert Morris, and M. Frans Kaashoek. Comparing the performance of distributed hash tables under churn. In *In The 3th International Workshop on Peer-to-Peer Systems (IPTPS'04)*, 2004.

[MAD02] Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal byzantine storage. In *DISC '02: Proceedings of the 16th International Conference on Distributed Computing*, pages 311–325, London, UK, 2002. Springer-Verlag.

[Maz01] David Mazières. A toolkit for User-Level file systems. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 261–274, Berkeley, CA, USA, 2001. USENIX Association.

[mBPS05] Jean michel Busca, Fabio Picconi, and Pierre Sens. Pastis: A highly-scalable multi-user peer-to-peer file system. In *in Euro-Par 2005*, 2005.

[MBRI03] Gurmeet S. Manku, Mayank Bawa, Prabhakar Raghavan, and Verity Inc. Symphony: Distributed hashing in a small world. In *In Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, pages 127–140, 2003.

[MCG05] M. Muhammad, A. S. Cheema, and I. Gupta. Efficient mutual exclusion in peer-to-peer systems. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, GRID '05, pages 296–299, Washington, DC, USA, 2005. IEEE Computer Society.

[MCM01] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 174–187, New York, NY, USA, 2001. ACM Press.

[MD88] P. Mockapetris and K. J. Dunlap. Development of the domain name system. *SIGCOMM Comput. Commun. Rev.*, 18:123–133, August 1988.

[MGGM04] Sergio Marti, Prasanna Ganesan, and Hector Garcia-Molina. Dht routing using social links. In *3rd International Workshop on Peer-to-Peer Systems (IPTPS 2004)*, February 2004.

[Mha11] Darshan Mhapasekar. Accomplishing anonymity in peer to peer network. In *Proceedings of the 2011 International Conference on Communication, Computing & Security*, ICCCS '11, pages 555–558, New York, NY, USA, 2011. ACM.

[MI09] Ana Lúcia De Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.*, 31:6:1–6:31, February 2009.

[MKKW99] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, SOSP '99, pages 124–139, New York, NY, USA, 1999. ACM.

[MM02] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric, 2002.

[MMGC02] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A Read/Write Peer-to-Peer file system. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation*, 2002.

[MNR02] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 183–192, New York, NY, USA, 2002. ACM.

[MR97] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 569–578, New York, NY, USA, 1997. ACM Press.

[MT03a] Tim Moreton and Andrew Twigg. Enforcing collaboration in peer-to-peer routing services. In *Proceedings of the 1st international conference on Trust management*, iTrust'03, pages 255–270, Berlin, Heidelberg, 2003. Springer-Verlag.

[MT03b] Tim Moreton and Andrew Twigg. Trading in trust, tokens and stamps. In *1st Workshop on the Economics of P2P systems*, 2003.

[NSN] http://www.nsnam.org.

[OM94] Kazuo Ohta and Mitsuru Matsui. Differential attack on message authentication codes. In *Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '93, pages 200–211, London, UK, 1994. Springer-Verlag.

[Omn] http://www.omnidrive.com.

[Ope] http://www.openomy.com.

[Osa88] Alex Osadzinski. The network file system (nfs). *Comput. Stand. Interfaces*, 8:45–48, July 1988.

[OSV09] Christian Ortolf, Christian Schindelhauer, and Arne Vater. Classifying peer-to-peer network coding schemes. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 310–318, New York, NY, USA, 2009. ACM.

[PCT04] Bogdan C. Popescu, Bruno Crispo, and Andrew S. Tanenbaum. Safe and private data sharing with turtle: Friends Team-Up and beat the system. In *In Proc. of the 12th Cambridge Intl. Workshop on Security Protocols*, 2004.

[PPD+95] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from bell labs. *Computing Systems*, 8(3):221–254, 1995.

[PRR97] Greg C. Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[PSAS01] Marius Portmann, Pipat Sookavatana, Sébastien Ardon, and Aruna Seneviratne. The cost of peer discovery and searching in the gnutella peer-to-peer file sharing protocol. In *Proceedings of the 9th IEEE International Conference on Networks*, ICON '01, pages 263–, Washington, DC, USA, 2001. IEEE Computer Society.

[RD01a] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for Large-Scale Peer-to-Peer systems. *Lecture Notes in Computer Science*, 2218:329–351, 2001.

[RD01b] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of the 18th ACM Symposium on Operating System Principles*, October 2001.

[RFSH01] Sylvia Ratnasamy, Paul Francis, Scott Shenker, and Mark Handley. A scalable Content-Addressable network. In *In Proceedings of ACM SIGCOMM*, pages 161–172, 2001.

[RGRK04] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *Proc. of the 2004 Usenix Annual Technical Conference*, June 2004.

[RH04] Sriram Ramabhadran and Joseph M. Hellerstein. Prefix hash tree: An indexing data structure over distributed hash tables, 2004.

[Sa07] Jeremy Stribling and Emil Sit and. Don't give up on distributed file systems. In *Proc. of the 6th IPTPS*, February 2007.

[SBA03] Sbarc: A supernode based peer-to-peer file sharing system. In *Proceedings of the Eighth IEEE International Symposium on Computers and Communications*, ISCC '03, pages 1053–, Washington, DC, USA, 2003. IEEE Computer Society.

[SFH+99] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. *SIGOPS Oper. Syst. Rev.*, 33(5):110–123, 1999.

[SKK+90] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[SKKM02] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the pangaea wide-area file system. *SIGOPS Oper. Syst. Rev.*, 36(SI):15–30, 2002.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, volume 31, pages 149–160, New York, NY, USA, October 2001. ACM.

[SMZ03] K. Sripanidkulchai, B. Maggs, and H. Zhang. Efficient content location using interest-based locality in peer-to-peer systems. In *IEEE Infocom*, San Francisco, CA, March 2003.

[SNDW06] Atul Singh, Tsuen-Wan Ngan, Peter Druschel, and Dan S. Wallach. Eclipse attacks on overlay networks: Threats and defenses. In *25th Conference on Computer Communications (INFOCOM 2006)*. IEEE, 2006.

[SS96] Mirjana Spasojevic and M. Satyanarayanan. An empirical study of a wide-area distributed file system. *ACM Trans. Comput. Syst.*, 14(2):200–222, May 1996.

[SS02] J. Sabater and C. Sierra. Regret: a reputation model for gregarious societies. In C. Castelfranchi and L. Johnson, editors, *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2002)*, pages 475–482. ACM Press, 2002.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, March 2005.

[Vas08] Nadejda Belbus Vasilyevna. An rbac design with discretionary and mandatory features. In *Proceedings of the 2008 International Symposium on Ubiquitous Multimedia Computing*, pages 260–263, Washington, DC, USA, 2008. IEEE Computer Society.

[Vog99] Werner Vogels. File system usage in windows NT 4.0. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 93–109, New York, NY, USA, 1999. ACM.

[WA93] Randolph Y. Wang and Thomas E. Anderson. xFS: A wide area mass storage file system. In *Workshop on Workstation Operating Systems*, pages 71–78, 1993.

[WDG+06] Charles P. Wright, Jay Dave, Puja Gupta, Harikesavan Krishnan, David P. Quigley, Erez Zadok, and Mohammad N. Zubair. Versatility and unix semantics in namespace unification. *Trans. Storage*, 2(1):74–105, 2006.

[Win] http://skydrive.live.com.

[WK02] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 328–338, 2002.

[Wua] http://www.wuala.com.

[WV03] Yao Wang and Julita Vassileva. Trust and reputation model in Peer-to-Peer networks. In *Third International Conference on Peer-to-Peer Computing (P2P'03)*, 2003.

[XDr] http://www.xdrive.com.

[YM02] Beverly Yang and Hector G. Molina. Improving search in Peer-to-Peer networks. In *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, Washington, DC, USA, 2002. IEEE Computer Society.

[YVGM04] B. Yang, P. Vinograd, and H. Garcia-Molina. Evaluating GUESS and non-forwarding peer-to-peer search. pages 209–218, 2004.

[ZH07] Runfang Zhou and Kai Hwang. PowerTrust: A robust and scalable reputation system for trusted Peer-to-Peer computing. *Transactions on Parallel and Distributed Systems*, 18(4):460–473, 2007.

[ZKJ01] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, UC Berkeley, April 2001.

[ZKW05] Chi Zhang, Arvind Krishnamurthy, and Olph Y. Wang. Brushwood: Distributed trees in peer-to-peer systems. In *In Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS'05*, pages 47–57, 2005.

[ZSJ06] Bo Zhu, Sanjeev Setia, and Sushil Jajodia. Providing witness anonymity in peer-to-peer systems. In *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, pages 6–16, New York, NY, USA, 2006. ACM.