E-Wolf: A Distributed Online Social Network

Eyal Kibbar
E-Wolf: A Distributed Online Social Network

Project Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Eyal Kibbar

Submitted to the Senate of the Technion — Israel Institute of Technology
Adar A 5774 Haifa Feb 2014
The project thesis was done under the supervision of Prof. Roy Friedman in the Computer Science Department.

The generous financial support of the Technion is gratefully acknowledged.
# Contents

List of Figures ................................................................. iii  
List of Algorithms ........................................................... v  
Abstract .................................................................................. 1  

1 Introduction ........................................................................... 3  

2 An Overview of E-Wolf ......................................................... 7  
  2.1 Top Level Design ............................................................ 7  
  2.2 Peer Discovery ............................................................... 9  
  2.3 Reliable Communication .................................................. 9  
  2.4 DHT ............................................................................. 10  
  2.4.1 Data Lifetime ............................................................ 10  
  2.5 ChunKeeper ................................................................. 11  
  2.5.1 The Storage Resource ............................................... 11  
  2.6 Stash ............................................................................ 12  
  2.7 Social File System .......................................................... 12  
  2.7.1 Consistency ............................................................... 13  
  2.8 E-Wolf ......................................................................... 13  

3 Communication Layer .......................................................... 15  
  3.1 Overview ....................................................................... 15  
  3.2 Related work ................................................................ 15  
  3.2.1 Crowds [31] .............................................................. 15  
  3.3 Key Based Routing ........................................................ 16  
  3.4 Distributed Hash Table (DHT) ......................................... 16  
  3.4.1 Limitations ............................................................... 16  
  3.5 Reliable Communication ............................................... 17  
  3.6 Conclusion .................................................................... 17  

4 Persistent Storage ................................................................. 19  
  4.1 Background ................................................................... 19  
  4.2 Related work ................................................................ 19  
  4.2.1 Ceph [38] ................................................................. 20  
  4.2.2 Farsite [4] ............................................................... 20
4.3 ChunKeeper .............................. 21
  4.3.1 Fair storage ....................... 22
  4.3.2 Protocol ......................... 23
  4.3.3 Interface ......................... 24
  4.3.4 Extensions ...................... 24
4.4 Stash .................................. 25
  4.4.1 Credentials ....................... 25
  4.4.2 Groups .......................... 26
4.5 Conclusion ............................ 27

5 Online Social Network .......... 29
  5.1 Background ......................... 29
  5.2 Related work ...................... 29
    5.2.1 SafeBook [13] .................. 29
    5.2.2 Confidant [27] ................. 30
    5.2.3 Contrail [35] .................. 32
    5.2.4 PeerSoN [8] .................... 33
    5.2.5 Diaspora ....................... 34
  5.3 SocialFS ............................ 35
    5.3.1 Architecture .................... 35
    5.3.2 Consistency ..................... 37
  5.4 E-Wolf .............................. 38
    5.4.1 Architecture .................... 38
  5.5 Conclusion ......................... 40

6 Kaleidoscope: Adding Colors to Kademlia[14] 41
  6.1 Abstract ............................ 41
  6.2 Introduction ....................... 41
  6.3 Related Work ....................... 43
  6.4 A Brief Overview Of Kademlia .... 44
  6.5 Kaleidoscope ....................... 45
    6.5.1 Colors .......................... 45
    6.5.2 The Routing Scheme .......... 45
    6.5.3 Overload Protection ........... 47
    6.5.4 Short Forwarding Phase ...... 49
    6.5.5 Discussion ..................... 50
  6.6 Analysis ............................ 50
    6.6.1 Expected Performance .......... 50
    6.6.2 Load Distribution Between the Colors ... 54
  6.7 Performance Measurements ....... 55
    6.7.1 Mathematical Analysis Evaluation . 56
    6.7.2 Performance Overview ......... 56
    6.7.3 Effect of Overload Protection ... 57
    6.7.4 Different Color Configurations .. 57
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.7.5 Kaleidoscope vs. Local and KadCache</td>
<td>58</td>
</tr>
<tr>
<td>6.8 Conclusions</td>
<td>59</td>
</tr>
<tr>
<td>7 Discussion</td>
<td>63</td>
</tr>
<tr>
<td>7.1 Future work</td>
<td>63</td>
</tr>
<tr>
<td>7.1.1 Communication Layer</td>
<td>63</td>
</tr>
<tr>
<td>7.1.2 Storage Layer</td>
<td>64</td>
</tr>
<tr>
<td>7.1.3 Social Network Layer</td>
<td>64</td>
</tr>
<tr>
<td>Bibliography</td>
<td>65</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>System Overview</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>System Entities Overview</td>
<td>9</td>
</tr>
<tr>
<td>4.1</td>
<td>Insert Group</td>
<td>27</td>
</tr>
<tr>
<td>5.1</td>
<td>SocialFS Files</td>
<td>36</td>
</tr>
<tr>
<td>5.2</td>
<td>SocialFS File hijacking</td>
<td>38</td>
</tr>
<tr>
<td>5.3</td>
<td>E-Wolf <em>wolf pack</em></td>
<td>40</td>
</tr>
<tr>
<td>6.1</td>
<td>A routing example for Kaleidoscope</td>
<td>48</td>
</tr>
<tr>
<td>6.2</td>
<td>Theory vs. experiment</td>
<td>57</td>
</tr>
<tr>
<td>6.3</td>
<td>Caching vs. no caching</td>
<td>58</td>
</tr>
<tr>
<td>6.4</td>
<td>Overload protection</td>
<td>59</td>
</tr>
<tr>
<td>6.5</td>
<td>Number of colors comparison</td>
<td>60</td>
</tr>
<tr>
<td>6.6</td>
<td>Caching schemes comparison</td>
<td>61</td>
</tr>
<tr>
<td>6.7</td>
<td>Normalized message cost table</td>
<td>61</td>
</tr>
<tr>
<td>6.8</td>
<td>Cache size comparison</td>
<td>62</td>
</tr>
</tbody>
</table>
List of Algorithms

4.1 ChunKeeper Store operation ........................................ 28
6.1 The Forwarding Scheme ............................................. 46
6.2 The Find Value Scheme ............................................. 47
Abstract

This thesis presents E-Wolf, a fully decentralized implementation of an on-line social network. E-Wolf follows a modular design in which the complex functionality of the system is divided into layered modules such that each module builds on the functionality of the ones below it. This design enabled gradual development of the system and favors extendability and maintainability.

More specifically, in E-Wolf, every node participates in a Peer to Peer network. The most basic E-Wolf modules support discovery and routing in such a network. On top of these, various storage mechanisms are provided, each suitable for different type of interactions needed for storing both internal management data and users’ generated data. These in turn are extended by the upper most E-Wolf modules to provide the known social networking experience. Also, all data is encrypted and an emphasis was placed on how users can control the privacy of their data and who has access to it. Further, an attempt is made to keep traffic local when possible.

E-Wolf is implemented in Java and is available as an open source project. In fact, each of its modules is packaged as a separate project. This enables using only parts of E-Wolf in other projects, and encourages developing additional modules providing additional functionality for the system.
1 Introduction

In recent years, online social networks have become an integral part of our lives. Many of us are enlisted not only to one but several different social networks. Each flavor of a social network focuses on different aspects of our lives and targets different kind of audience and usage patterns. LinkedIn\textsuperscript{1} focuses on the professional point of view. Users there tend to publish information related to their occupation and their skills. Thus, much consideration is given to what they publish and with whom they share it. Facebook\textsuperscript{2} on the other hand, is more private life oriented. Users publish photos from their social life and share it with friends and family.

Depending on the data users post and with whom they share it, online social networks use different privacy definitions and sharing modules. While friendship in Facebook is only defined symmetrically, that is, my friends can see everything I publish on my profile and vice-versa, other social networks have a more asymmetric approach. For instance, Twitter uses a publish subscribe model. Each user is following other users, whom he can see all their twits, and has followers who can see his twits. This different notion of sharing made twitter more suitable for celebrities keeping in touch with their fans than Facebook. Other models have more fine grain sharing policies. For example, in Whatsapp, each post is shared with a known group of contacts allowing the users better control over who sees the post. This property makes Whatsapp\textsuperscript{3} ideal for arranging meetings of a medium size groups who share common interest, such as mothers who take their children to play together in the local park. Using the wrong permission to a post can have significant and sometimes deadly influences as can be understood from the many cases of publicly available birthday party invitations on Facebook that ended up in a riot of several thousands people. The business model of these social networks often prevents them from treating the privacy of their users as a major design goal.

Further, each of the popular online social networks hosts hundreds of millions of users. For example, Facebook’s third quarter 2013 reports indicated that Facebook alone holds 1.19 Billion monthly active users, 728 Million out of them are daily active users. This makes the data-centers of online social networks prime targets for potential infiltrators. Further, supporting such a large number of users by a single company incurs tremendous operational costs, which serve as a major barrier of entry for potential competitors.

\textsuperscript{1}https://www.linkedin.com/
\textsuperscript{2}https://www.facebook.com/
\textsuperscript{3}http://www.whatsapp.com/
In the past, there were several attempts to design decentralized online social network, which aim to alleviate the limitations of centralized ones, namely the huge operational cost, the temptation to sacrifice users’ privacy in order to reduce costs and increase profit, and being prime targets for infiltration. In Diaspora\textsuperscript{4}, for example, the designers attempt to separate the physical infrastructure (i.e. storage and CPU providers) from the service. They strive to define a standard protocol for social interaction, by which they hope to encourage a large variety of small independent infrastructure providers. These infrastructure providers will participate in a large decentralized network communicating with each other using the standard social protocol and providing access to the social network for their customers. Another approach, such as the one presented in [13], uses the customers themselves to create the social network. In this approach, every participating user contributes some of his resources to the social network. These approaches demonstrate a trade-off between privacy and complexity. While the former approach is rather simple to maintain and to enforce sharing permissions, it requires the exposure of the user’s data before the service provider. The latter, on the other hand, is much more privacy friendly. However, it requires sophisticated mechanisms in order to ensure the same availability, performance and permissions enforcements as the former approach.

In this thesis, we describe E-Wolf, a decentralized social network that supports remote access from both donated (home) computers and mobile devices. To that end, in E-Wolf we distinguish between computers that form and host the network, called \textit{nodes}, the devices (mobile or stationary) used to access the network, called \textit{clients}, and \textit{users}, who are the people using the network through client devices and who donate the nodes to the system.\textsuperscript{5} In particular, all nodes are equal and there are no providers and consumers or special roles of any kind to a certain subset of the network. Every node must contribute resources (CPU, storage, bandwidth) to the social network and there are mechanisms to ensure the fair share of each user.

As for the information sharing model, we strive to provide a clear yet powerful control over the distributed data. Therefore, we have adopted the per-post permissions sharing model. Each post is shared with a selected subset of the user’s contacts. Since each post is shared by a weakly connected set of users, every addition to the original post must be approved by the post creator, who is the common grounds for all of them.

The architecture of E-Wolf is modular. That is, in designing E-Wolf, we have identified composable functionality building blocks and designated a specific module for each of them. Each such module is developed as a stand-alone component with well specified interface that only depends on the functional behavior (and interface signature) of the modules below it. Together, they provide an implementation of an online social network. Yet, each of these modules is presented as an open source

\textsuperscript{4}https://joindiaspora.com/

\textsuperscript{5}The same physical computer may act as both a node and a client device. However, typically mobile phones and tablets only act as clients.
project and indeed the lowest layer of E-Wolf has already been downloaded and used multiple times by third party projects.

The rest of this thesis is organized as follows: In Chapter 2 we give an overview of E-Wolf’s design. In Chapter 3 we describe the basic communication algorithms provided in E-Wolf. In Chapter 4 we present the storage mechanism E-Wolf uses. In Chapter 5 we elaborate on the social interactions E-Wolf provides. In Chapter 6 we describe Kaleidoscope [14], E-Wolf’s P2P network. Finally, we conclude with a discussion in Chapter 7, where we suggest future work for the development of E-Wolf.
2 An Overview of E-Wolf

In this chapter, we discuss and define key features of E-Wolf architecture. As mentioned in the introduction, E-Wolf is an implementation of a privacy aware distributed online social network. It is currently implemented in Java. However, an E-Wolf node can be written in any language as long as it complies to all the protocols that E-Wolf uses.

2.1 Top Level Design

E-Wolf is composed of several layers, each of which has its own responsibilities and can be considered as a standalone project detached from the implementation details of layers above or below it. The layers of E-Wolf can be categorized into four different roles:

1. Communication layer - responsible for low level communication between peers. In this level we provide peers with the means to send messages to each other as well as finding each other. Messages can be sent reliably or unreliably and be polled or pushed. In this layer we can find the KBR DHT and HTTP connector components, which are discussed thoroughly in Chapter 3.

2. Storage layer - responsible for storing persistent data. The task of storing data in a peer to peer environment has always been difficult since storage is a limited resource and peers must be encouraged to cooperate with each other in an untrusted world. To make matters worse, storage is the root of another limited resource: Bandwidth. The more data one stores, the greater the probability that some of this data would be interesting for other peers, thereby increasing the required up stream bandwidth. Thus, providing fair distribution of this resource is crucial for the success of E-Wolf. We can separate the storage task into two distinct sub tasks:

   a) Replicate as required and discard as needed with fairness in mind. Data is replicated in order to make it as available as possible. When different peers hold the same data it can potentially increase the percentage of time in which the data exists in at least one online peer. Furthermore, replicas can be used to download the data from multiple sources, thus decreasing data arrival latency. Another challenge we have with replication is garbage collection. Data may become unreachable for various
reasons, such as storage failures or corruption, and we must take it into
account. Unreachable data must be eventually removed and unpopular
data should decrease the number of its replicas. This task is implemented
by the ChunKeeper component discussed in Chapter 4.

b) Encryption and signatures. In a privacy aware social network, all data
must be encrypted and signed in order to prevent untrusted peers from
reading or changing it. One very important data we must encrypt, for
example, is the credentials file. This file holds all the user’s secret keys
which are used in the storage layer and all layers above it. This task is
implemented by the Stash component discussed also in Chapter 4.

3. File system abstraction - responsible for creating an easy to use, familiar ab-
trraction of a file system. The social file system also provides each user with
a public profile where he can publish his name, picture and any public data
he wishes to share with the rest of the world. One important aspect of a file
system is its tree structure. Using this structure we can easily traverse all the
user’s data and delete unreachable data for increasing the user’s quota. This
task is implemented by the SocialIFS component discussed in Chapter 5.

4. Social Network - provides the user with all social network services: share posts,
send messages and comment on others’ posts. This layer is implemented by
the E-Wolf component discussed in Chapter 5.

(Figure 2.1) displays each component and its dependencies. The colors indicate the
layer categorization. Figure 2.2 displays the different entities in each layer.

![System Overview Diagram]

**Figure 2.1**: System Overview
2.2 Peer Discovery

Peer discovery is the problem of finding peers matching a given ID in a peer to peer network. In this task, we adopt a very promiscuous approach: if no peer with the desired ID is found, several peers with close IDs, according to some metric, are returned. This slack allows us to assign changing semantics of a peer’s role in the system. For example, we can have the following invariant: the peer with the ID closest to \( k \) holds the data \( d \), where \( k \) is a fixed size key that is generated out of a set of \( d \)'s properties. In order to enforce this invariant, the peer holding data \( d \) will do the following procedure:

1. Search the network for peers with ID equals to \( k \)
2. Once several peers with IDs closest to \( k \) are found, send \( d \) to the closest one
3. Wait some time and repeat step 1

The node having an ID closest to \( k \) may change from time to time as peers join and leave the network. However, our invariant will hold. As long as the peer who has data \( d \) follows this procedure, the data \( d \) will be sent to the node with the closest ID to \( k \).

In general, this routing approach is called *Key Based Routing (KBR)*. This is the basis for all communication in E-Wolf.

2.3 Reliable Communication

Providing reliable communication between peers is important for transferring large data, such as images and videos. However, reliable communication is expensive. Opening a TCP connection requires a 3-way handshake which can consume a lot
of time if done too frequently. The reliable communication layer (HTTP Communication) can reduce the overhead caused by starting new connections simply by caching and reusing open connections. It can also provide configurable anonymity layer using proxy techniques similar to [31].

2.4 DHT

DHT stands for Distributed Hash Table. It provides a key/value lookup semantics. The DHT heavily relies on the KBR infrastructure. Inserting data is done using peer lookup for the data’s key and sending the data to the \( k \) closest peers. Data retrieval is done by querying all the found peers closest to the requested key. Data in the DHT is not trusted and all messages are sent unreliably, thus data insertion and retrieval can only be probabilistically guaranteed. Moreover, data may get lost due to a variety of node failures, such as network disconnection or shutdown. E-Wolf uses three different DHTs all running on the same KBR but with different data semantics and handling.

1. **Chunks DHT** - Provides a map between chunk id and its location (IP and port of the chunk holder). Data in this DHT is only re-inserted by the node that generated it and has a short life time.

2. **Profiles DHT** - Holds the profile information, such as name and ID of all users. Data in this DHT is re-inserted periodically by every node currently holding it. In order to avoid over replication, the holding node may remove the data if it is not among the target nodes. The re-insertion process prolongs the data availability significantly since it keeps the replication factor high compensating for nodes that went offline. Data in this DHT has long life time limit allowing its origin a long period between successive re-insertions. Note that profiles are only persistently saved by the user whom the profile belongs to, thus staying offline for a long time period may result in the user’s profile getting lost rendering him unfindable until his next login.

3. **Messages DHT** - Used to pass messages between users. As in the Profiles DHT, data in this DHT is also re-inserted periodically. It has a medium life time (around 7 days) allowing users to send messages to each other even when only one user is online.

2.4.1 Data Lifetime

For every piece of data found in the DHT there are two entities involved:

1. The data producer - the peer who created the data and was the first peer to insert it. This peer can possibly recreate the data should it be lost for some reason.
2. The data holders - these peers are a small group who’s IDs happened to be close to the data’s key according to the KBR and have received a store request for the data. When data is replicated by both the data’s producer and holders, it can survive after the producer leaves the network. However, since the DHT has only probabilistic guarantee on data transfer, every piece of data replicated solely by the data holders has an expected maximum life time before it cannot be found by anyone. The lifetime parameters place an upper bound for the data validity period where beyond it the data is considered stale and will not be replicated anymore by the data holders. A good example of data that can get stale very quickly is the IP address and port of a peer. Such data is irrelevant once the peer has left the network.

2.5 ChunKeeper

While the DHT stores volatile and small data, ChunKeeper supplies persistent storage for any size of data. Data in E-Wolf can be categorized into two completely different mutually exclusive types:

1. User data - All data inserted by the human user, such as pictures, comments, videos, secret keys etc. User data is mostly characterized by being persistent with long lifetime, although there are some exceptions that will be discussed later.

2. Management data - Data used by E-Wolf, which the human user has neither direct knowledge about nor any control over it. This type of data is characterized by being volatile with short lifetime and cannot be associated with any particular user. Among the many items that fall under this category are the list of known peers in the network and addresses of peers holding some user data.

ChunKeeper is mostly used for holding user’s data. For this reason, it has a fairness mechanism built into it. Whereas management data is small and tends to distribute uniformly across peers, user data tends to distribute in a non-uniform manner. Some users have lots of pictures, others upload only textual posts. Some users have lots of friends and are subscribed to many groups, others have only a few friends and are only subscribed to a few groups. Thus, a fairness mechanism must be implemented in order to encourage users to donate as many resources as they can and punish those who exceed their limit.

2.5.1 The Storage Resource

In recent years, persistent storage devices have grown in their capacity exponentially. This trend has left many of us with a huge amount of unused storage space. Even
though limiting this resource and enforcing fair share of it seems pointless, we argue that it is the root cause of all other limited resources in the system. The more data you store, the greater the chance some will need a part of your stored data, resulting in increased upload bandwidth consumed by sending it. The CPU resource is also affected by the storage one. Aside from handling more TCP connections due to a greater number of requests as discussed above, peers may ask for data being compressed or encrypted before being sent. Needless to say, encryption and compression are very CPU intensive tasks.

Therefore, ChunKeeper fairness mechanism only tracks storage usage and its availability and encourages users to both share as much as they can and stay online as long as they can.

### 2.6 Stash

After having a persistent and fair storage, we would like to use it for storing users private data. However, storing private data on untrusted machines requires it to be encrypted. Stash is a thin layer providing encryption and signatures for all data stored in ChunKeeper. It defines a user credentials were all private keys and symmetric secret keys are stored. Any additional secret key is stored in ChunKeeper encrypted using the master secret key in the user credentials under the key built using a cryptographic hash function on the following values:

1. Random number defined in the user credentials for preventing two users from having the same key.
2. A sequence number for changing the key value for different secret keys.

The user credentials is itself stored in ChunKeeper and encrypted under a user chosen password.

Whenever dealing with passwords chosen by human users, we must always take into account the fact that humans tend to choose simple passwords such as a word or a name. Once a user’s credentials is obtained, one can try to guess its different passwords until he succeeds. Therefore, Stash implements another mechanism that prevents these offline brute force attack on the credential file. Further details are discussed in Chapter 4.

### 2.7 Social File System

In order to manage all the user data in an accessible manner we need another abstraction. File systems provide such abstractions, as their tree structure allows us to traverse all our data recursively. This easy access is used for two crucial periodical operations:
1. Garbage collections - from time to time, data can be lost due to insufficient replication, corruption or simple lose of password. The data that was lost can contain pointers to other data items. One such example is a folder in a file system. Losing a folder can render all its sub-folders and files inaccessible. Using the simple tree structure, we implement a Mark & Sweep algorithm to delete these files.

2. Exporting data - once in a while a user might want to backup his data. The tree structure allows downloading all the accessible user’s data.

Furthermore, file systems must provide some synchronization mechanism between file and folder creation and modification. Needless to say that in every distributed system, the issue of synchronization complexity greatly affects both the system’s consistency and its performance. In most systems there is a trade-off between consistency to either performance or features. Thus, figuring out the correct model of synchronization must take into account the way we use our file system in order to reduce as many unnecessary features as possible for tipping the balance towards achieving better performance without compromising the consistency.

### 2.7.1 Consistency

Unlike traditional file systems, data in SocialFS has a built-in strong consistency mechanism. Since SocialFS uses ChunKeeper, it is guaranteed that every chunk is a consistent, properly ordered autonomous unit of data. Therefore, the only consistency issue in SocialFS is the proper ordering of different chunks in folders, whereas the consistency of each piece of data, such as a post, a photo, a video, etc, is handled by ChunKeeper. SocialFS orders every piece of data by its creation date according to ChunKeeper. However, if two items are inserted to SocialFS simultaneously by the same user, there is no guarantee about the order in which these items will be displayed in the future. For example, if a user posts two posts simultaneously, each peer that tries to view them may see different ordering of the posts. This inconsistency not only has a minor affect on the overall experience of social networking but also is extremely rare. Moreover, it allows us to greatly increase our performance since we do not require any locking or consensus [25] of any kind. Note that even though SocialFS does not guarantee sequential consistency, it does guarantee causal consistency [24].

### 2.8 E-Wolf

The last component that puts everything together is E-Wolf. This component defines all the activities allowed by users and organizes the data in SocialFS. E-Wolf allows users to post images, text and videos on their walls, share these posts with other peers and comment on others’ posts. For each user, we define a personal space where
he can add posts and a message box for sending and receiving messages from other users. The user is the only one who can add posts and comments to his personal space. Thus, whenever he wishes to comment on another user’s post, he must first send a comment request. Then the post’s owner must approve the comment before it is added to his post. This approach allows full control over the comments’ content in order to prevent spam.

E-Wolf implements a per-post read permission. That is, each post is encrypted using a group key. When the user Alice decides to add Bob to a group, she sends a message containing the group ID and a secret key she wishes to add Bob to. For example, Alice might have a group named “family”. Every post shared with the group “family” is encrypted with the “family” group’s secret key. Every person Alice had added to the “family” group had been sent Alice’s “family” group secret key. If he approved Alice’s invitation, the secret key is saved under Alice’s “family” group ID in his personal storage place, provided by ChunKeeper.
3 Communication Layer

3.1 Overview

E-Wolf’s communication layer provides several means of node communication for the layers above it. Each layer in E-Wolf requires a somewhat different abstraction for its communication. For instance, ChunKeeper requires both reliable communication for transferring large chunks between storing nodes and small message passing means for meta-data, such as chunks location. In contrast, E-Wolf instant text messaging mechanism only requires small unreliable message passing.

E-Wolf communication layer is composed of 3 modules. Each differ in the services it provides, the maximum size of message they can carry and the reliability of the transfer itself:

1. Key based routing network - Implements a variant of Kademlia [30]. It provides peer discovery for other modules.
2. Distributed hash table - A thin layer for enabling several data semantics to co-exists on the same key based routing network.
3. Reliable communication - A communication layer for transferring large amount of data reliably between nodes.

3.2 Related work

3.2.1 Crowds [31]

Crowds provides client anonymity by implementing a “blend into a crowd” approach, i.e., hiding one’s actions within the actions of many others. To execute Web transactions, a user first joins a “crowd” of other users. The user’s request to a web server is first passed to a random member of the crowd. That member can either submit the request directly to the end server or forward it to another randomly chosen member. In the latter case, the next member chooses to submit or forward independently. When the request is eventually submitted, it is submitted by a random member, thus preventing the end server from identifying the initiator of the request, since the initiator is indistinguishable from the member that simply forwards the request on behalf of another.
Chapter 3 Communication Layer

3.3 Key Based Routing

The key based routing module is the very core of E-Wolf communication layer. It provides us a peer discovery service and a simple message passing between nodes. E-Wolf uses a modified implementation of Kademlia called Kaleidoscope [14]. Kaleidoscope provides better performance than Kademlia by caching peer discovery operations results.

3.4 Distributed Hash Table (DHT)

The DHT module heavily relies on the Key Based Routing module. It adds a storage layer with semantics for items replication and lifetime. As in a regular hash table, the interface is a simple get and put operations. The DHT module defines two request messages that all nodes must handle and may send:

1. Store request - A request to store a new item in the storage. The request specifies both key and item’s content.

2. Get request - A request to send back the item’s content. The request only specifies the item’s key.

The DHT module defines a maximum lifetime for each inserted item beyond which the item will be deleted from the storage. This property is suitable for volatile rapidly changing items such as IP addresses of nodes. Another ability of the DHT is periodic items re-insertion. Each configurable amount of time, the DHT performs a peer discovery operation to all the keys in its storage and sends a Store request to all the found nodes. This property is useful when trying to keep small items available for as long as possible without paying the overhead of finding persistent storage for these small items. For instance, text messages fit this category perfectly. We desire them to be available even when the message sender is offline. However, using ChunKeeper for such small and short lifetime messages may cause a great overhead in ChunKeeper management which over time may degrade its performance.

A great advantage we have achieved by separating the Key Based Routing from the DHT module was the ability to have several DHTs use the same Key Based Routing network for their underlaying peer discovery operations. E-Wolf heavily relies on this ability were it has three different DHTs with different configurations, which is discussed later in Chapters 4 and 5.

3.4.1 Limitations

The DHT module has several limitations due to its implementation and the Key Based Routing it uses as an infrastructure for sending messages between peers.

---

1https://code.google.com/p/dht/
3.5 Reliable Communication

1. Message passing is unreliable since we use the Key Based Routing message passing mechanism.

2. Messages have limited size. Since each message is encapsulated as a UDP packet, we have a theoretical limit of 64KB. However, the practical limit is much lower due to fragmenting a UDP packet according to the MTU size, which significantly decreases the chances of its arrival. Thus, the practical limit is only 1500B.

3.5 Reliable Communication

The reliable communication module allows nodes to transfer large amounts of data reliably. It uses the well-known and widespread protocol of HTTP. As defined in RFC 2068, HTTP supports proxy connections. This allows us to obfuscate communication using several proxy hops before communication with the destination node is established. By skipping several hops we provide the client with anonymity from the destination node. In order to control the number of hops, we use two configurable parameters very similar to the technique presented in Crowds [31]:

1. Paranoia - The chance of a request generated by the local node to be sent to a proxy instead of the destination node.

2. Hysteria - The chance of a received request from another node to be sent to yet another proxy instead of the destination node.

By setting Paranoia to a high value, we can obfuscate our own requests. However, we degrade the performance since our requests tend to go through proxies. The Hysteria parameter on the other hand, is more altruistic. As in Paranoia, setting a high value to this parameter degrades the overall performance of the network. However, it has little to no affect on the local node requests performance. It is used merely to increase the overall obfuscation in the entire network rather than to help individual obfuscation.

Proxy selection is currently implemented by selecting a random node from the known nodes list of the Key Based Routing network. However, future implementation might be integrated with the layers above making it possible to select a proxy from a list of trusted nodes or even rely on social connections for trust as suggested in SafeBook [13].

3.6 Conclusion

As we have seen, E-Wolf communication layer provides anonymity at its very core. This anonymity is achieved using several different techniques. In the Key Based Routing level, caching peer discovery operation results is used to hide the keys we
are searching. In the reliable communication level, proxy connection are used for hiding the nodes we are connecting from. However, the most important feature of this design is its modularity. Each module can be replaced by an improved version of itself while the other modules and the layers above it stay oblivious to the change.
4 Persistent Storage

In this chapter, we discuss ChunKeeper and Stash. Together, these two systems provide a reliable, scalable and trusted persistent storage for the E-Wolf social network. We also discuss numerous other distributed storage systems and see that each one has a significant drawback rendering it unfit for E-Wolf.

4.1 Background

A distributed file system is inherently different from a local one. When splitting chunks of data over different nodes we must take into account new factors. We must assume that both nodes cannot be trusted and communication may be disrupted or recorded by an attacker.

Untrusted communication has several implications. An attacker might eaves drop to our communication or even disrupt it in such a way that we end up with false data. The very existence of a connection between two nodes may very well be important. For instance, say an attacker has discovered a certain individual that has top secret file on its node. Even though the attacker does not know the contents of the file, it is still very interesting to know who the file was sent to.

Untrusted nodes may change the data, delete it, or refuse to send it. In any large distributed system there must be a mechanism for encouraging nodes to behave nicely and share their resources either by rewarding the good ones, punishing the bad ones or a combination of both. In order to prevent data leaking and violating privacy, all data must be encrypted and signed before it is sent to any node.

4.2 Related work

Many distributed file systems have been proposed in the past. Here we will discuss only a selected few. Note that each and every one of the distributed file systems we discuss makes the assumptions above (untrusted network and nodes) to some degree.
4.2.1 Ceph [38]

Ceph is a distributed file system that provides excellent performance and reliability while promising unparalleled scalability. Ceph’s architecture is based on the assumption that systems at the petabyte scale are inherently dynamic: large systems are inevitably built incrementally, node failures are the norm rather than the exception, and the quality and character of workloads are constantly shifting over time.

4.2.1.1 Architecture

The Ceph file system has three main components: the client, each instance of which exposes a near-POSIX file system interface to a host or process; a cluster of object storage devices, which collectively stores all data and metadata; and a metadata server cluster, which manages the namespace (file names and directories) while coordinating security, consistency and coherence.

4.2.1.2 Discussion

Ceph’s architecture shows us that creating a social network where users share files according to groups in one huge mountable file system is feasible. However, this design is more appropriate to a commercial service provider with the resources of acquiring and maintaining a cluster or a private cloud. Needless to say, this approach does not fit our goal since data is stored in one place unprotected and considerable resources by a central entity are needed in order to keep it running.

4.2.2 Farsite [4]

Farsite is a serverless distributed file system that logically functions as a centralized file server but whose physical realization is dispersed among a network of untrusted desktop workstations. Farsite is intended to provide both the benefits of a central file server (a shared namespace, location-transparent access, and reliable data storage) and the benefits of local desktop file systems (low cost, privacy from nosy sysadmins, and resistance to geographically localized faults). Farsite replaces the physical security of a server in a locked room with the virtual security of cryptography, randomized replication, and Byzantine fault-tolerance.

4.2.2.1 Architecture

Every machine in Farsite may perform the roles of a client and a member of a directory group. A client is a machine that directly interacts with a user. A directory group is a set of machines that collectively manage file information using a Byzantine-fault-tolerant protocol: Every member of the group stores a replica of
the information, and as the group receives client requests, each member processes
these requests deterministically, updates its replica, and sends replies to the client.
The Byzantine protocol guarantees data consistency as long as fewer than a third
of the machines misbehave.

4.2.2.2 Discussion

Even though Farsite seems a better choice than Ceph, it has several major disad-
vantages:

1. The directory group must have low-latency high-bandwidth communication
   within the group. This is due to the usage of a fault-tolerant protocol.
2. A majority of the directory group must be available most of the time.
3. Write privileges are enforced by the directory group.
4. Read permissions are enforced by re-replicating the file for each member who
   has read access and encrypting it for his public key.

Although the idea of having a selected small relatively trusted group (with at least
2/3 of the group acting honorably) can scale very good, it has major flaws when
using the Internet rather than a small inter-university or corporation network. Due
to the treacherous nature of the Internet environment, in E-Wolf we must trust the
least amount of peers as possible, assume long-latency low-bandwidth connections
and long downtime of peers.

4.3 ChunKeeper

Data in any storage system can be categorized by the following parameters:

1. User data or Meta-data. User data is the files, pictures and documents the
   user sees, modifies and reads. Meta-data is generated by the system in order
to find the user data when requested.
2. Persistent or volatile. Data may be both persistent and immutable, such as
   pictures. Volatile data has a short lifetime and may be updated frequently.
   For example, the IP of a node that holds a particular user data is volatile since
an IP address may change whenever the node disconnects.
3. Public, private or restricted. Public data can be viewed by all users; thus, it is
   not encrypted. However, it should still be signed for authentication. Private
data can be seen by only one user. Restricted data can be seen by a well
defined group of users. This sort of access is the most complicated to enforce
since it requires a secure key sharing mechanism.

1https://code.google.com/p/chunkeeper/
ChunKeeper is intended to store immutable user data. It has no notion of an access control layer. In other words, all chunks in ChunKeeper can be downloaded by anyone who desires them. The entire access control and verification mechanism is handled in the layer above it called Stash.

### 4.3.1 Fair storage

As discussed before, in order to provide high availability of data stored in ChunKeeper, we must replicated it over several nodes. The replication factor is depended on the age of the data. Old data tends to be less accessed in a social network. Thus, we can replicate it on fewer nodes. On the other hand, new data tends to be more popular. Thus, in order to provide a good quality of service and high availability, ChunKeeper tends to use a higher replication factor.

In order to prevent free-loaders who store lots of data in other nodes but do not participate in the collective effort, ChunKeeper creates a peer storage relationship. According to previous interactions with different peers each node decides whether to store their data or not. Such a peer may even decide to delete other peers data when their ratings drop or its capacity is depleting. Each peer $p$ it interacts with gets a rating depending on several factors:

1. Availability - The amount of time the other peer is online and ready to serve.

2. Quality - The amount of time it returned corrupted data. If the other peer holds chunks that $p$ also holds, $p$ may challenge the other peer by asking a hash of any portion of the data. To prevent denial of service attacks, only the owner of the data may challenge other peers.

3. Quantity balance - Defined as the difference between the amount of data owned by the peer that $p$ holds and the amount of the data $p$ owns that is held by the other peer. The more data $p$ store on the other peer, the more $p$ is likely to accept its data.

4. User’s data usage profile - The expected extra bandwidth the new data would generate. Some users may create more popular data than others. In order to keep the popular data available, we strive to distribute it as much as possible between different peers. Thus, the more bandwidth $p$ expect the new chunk will generate, the less likely $p$ is to accept it. The user’s data usage profile is calculated according to the user’s chunks upload statistics.

Using a combination of these factors, node $p$ can both decide whether to accept a STORE request as well as find a suitable peer that is likely to accept $p$’s STORE requests. This creates an “I owe you and you owe me” relationship between pairs of peers. The strength of this system is that the relationships are created automatically and arbitrarily. The nodes have no notion of either the user or any of its relationship with any other nodes.
4.3 ChunKeeper

4.3.1.1 Garbage Collection

In any large storage system, garbage data is always bound to appear. These pieces of data can be either corrupted, unused or lost due to no links to it. When designing a storage system, dealing with these pieces must be taken into consideration. Otherwise, as time passes, it will accumulate and degrade performance by overloading the DHT causing longer queries and consuming storage place on peers.

A great advantage of the fairness mechanism is its built in garbage collection. When users stop logging in to their accounts for any number of reasons, their data has an expiration date. As time passes, the better the chance of their chunks being deleted and not replicated. This is caused by the decline of their availability factor followed by a decrease of their overall rating, placing their chunks first in line for deletion when storage space is about to deplete. Furthermore, social networks tend to generate a lot of old unimportant data such as old posts and their comments. The fairness mechanism automatically reduces the replication level of these old posts.

4.3.2 Protocol

ChunKeeper has three types of requests:

1. STORE - Requests the remote node to store a chunk in its storage. Aside from the data, the chunk also includes the key it is published under for later queries. The remote node may either accept the request and store the data or decline. When storing the data, the remote node also records the ID of the requesting node for later rating calculations. In order to prevent forgery, store requests are signed using the node’s public key. The request includes the IDs of the sender and the receiver in order to prevent replay attacks to both nodes.

2. GET - Requests a chunk from the remote node. The request includes the chunk ID, which is the key it is published under, its size and several hash results on the data itself. The remote node may either send the data or notify that it does not have it.

3. CHALLENGE - Requests a hash of a portion of the chunk. The challenge may only be issued by the node that sent the STORE request for the data in hand. Any other node may be ignored. The portion of the requested hash may change in order to prevent pre-calculation of the challenge and delete the data later. It is intended to verify that the remote node still holds the data and not only its hash.

In order to find the stored data, ChunKeeper generates a meta-data structure called “chunk descriptor”. For each chunk that ChunKeeper stores, it publishes a chunk descriptor in the chunks DHT. The chunk descriptor the contains following information:

1. SHA-256 digest of the data.
2. The total size of the data.

3. Contact information of the node currently storing the data: IP, port and known protocols, such as HTTP.

ChunKeeper inserts all its chunk descriptors to the chunks DHT periodically. Each chunk descriptor is inserted under the same key as the data it represents.

Rating calculation is also done periodically. Once an hour each node $p$ goes over all the chunks it holds and sends a CHALLENGE request to a random subset of their owners. $p$ then records these interactions locally and if necessary, updates the node’s rating.

4.3.3 Interface

ChunKeeper defines a simple map-like interface. It has three different operations:

1. Store - Stores arbitrary data under a given key. The key length is 20 bytes and has no limitations on its content. Layers above ChunKeeper, such as SocialFS, use SHA-256 of several semantic tags that describe the data they store. The details are discussed later in Chapter 5.

2. FindChunk - Returns a set of chunk descriptors that match a given key. Chunk descriptor holds information about the location of the data that was stored under the given key.

3. DownloadChunk - Using the chunk descriptor, downloads the arbitrary data from the remote nodes. HTTP Communication is used for the download process.

While FindChunk and DownloadChunk operations are pretty strait forward, the Store operation is a little more complicated. A complete pseudo code of it can be found in Algorithm 4.1

4.3.4 Extensions

4.3.4.1 Anonymity

Although STORE requests cannot be anonymous, the GET request may very well be. This anonymity has two dimensions: protection against untrusted nodes to know I am holding a certain chunk and protection against untrusted nodes to know I am looking for a certain chunk. The first is solved using a simple proxy. Instead of publishing the chunk descriptor with my node’s details in it, we simply publish another trusted node’s details and request that node to act as proxy for our GET requests. Note that CHALLENGE requests need not pass through the proxy since nodes that may send a CHALLENGE request already know that we have the chunk. Preventing others from knowing what I am looking for is done in a very similar way.
4.4 Stash

Instead of looking for the chunk myself, I can ask a trusted node to do it for me and send me back the results.

4.3.4.2 Trust

ChunKeeper can be equipped with a trust mechanism. Instead of relying on direct interaction with peers, we can use a trust mechanism to query other peers who had interactions with the candidate peer we desire to store our data in. Other nodes may provide valuable information. However, they may also lie and provide false information. By gathering this information, we can calculate a more robust rating for that peer. One such way is ignoring the extreme ratings of all the peers we asked and using the average ratings of all the rest. This simple trust mechanism assumes most of the nodes are trustworthy and only a few lie. These few try to change the rating as much as possible, either to the good or bad scale. Ignoring them will leave us with only the true ratings and averaging them will give us the best estimation. A more detailed version of this mechanism is presented in [22].

Other more complex trust mechanisms can be used, such as relaying on social connections for trust. However, social connections have little to do with machine trust. For instance, I may have a friend whom I trust on the personal level. However, his computer is infected with a virus that may cause his node to act in an untrustworthy manner.

4.4 Stash

Stash is an Access Control Layer (ACL). It manages the read/write permissions for users of the E-Wolf social network. Stash provides author authentication using public key signatures as well as enforcing read permissions by encrypting the chunks.

4.4.1 Credentials

Every user in the E-Wolf social network has his own user credentials. This data is used for holding different user related information such as secret keys and special chunks IDs. The user credentials is itself a chunk saved in ChunKeeper and encrypted using Stash. Note that ChunKeeper has no notion of this special chunk, it seems like every other chunk in the system. This property adds another level of security: The user credentials is a quality target. Decrypting it can give an attacker control over the victim’s account. When encrypted, this chunk looks like every other encrypted chunk in the system, allowing it to hide in the haystack of all other chunks. Nevertheless, it is still possible to identify the credential chunks according to access patterns. These chunks tend to be less accessed since the owner probably has his own copy of it cached or stored locally.
Chapter 4 Persistent Storage

Stash encrypts the user credentials using the user’s own chosen password. Since users tend to select simple passwords, another mechanism was designed for protection against an offline dictionary attack:

1. When decrypted, the credentials hold no indication of validity. That is, if an attacker has found a credential chunk and decrypted it with some password, there is no way of distinguishing a correct decryption of the credential chunk from a wrong one, or even a wrong decryption of an arbitrary data chunk.

2. Credential chunks are mapped in ChunKeeper under a key composed of both the user’s password and another random number generated during the installation of E-Wolf. This random number has several roles. First, it acts as salt, preventing two users who happen to have the same password from knowing about it. Second, it helps hiding the credential chunk, since finding it requires guessing not only the user’s password but also this random number. Third, it creates a more unified distribution of keys making the chunk descriptors meta-data distribute more evenly among nodes in the chunks DHT.

Nevertheless, it is important to have a strong password since the credential chunk may be identified by other means, such as network access patterns. If an attacker identifies a node as it goes online, he may assume that one of the first FindChunk operations it does will be to get its credentials.

Stash uses only one field in the user credentials: the “groups master key”. This key is used to encrypt the user’s groups which is discussed in the following section.

4.4.2 Groups

Stash uses groups for encrypting chunks. Each group has its ID and a special random key used for encrypting all the chunks that belong to their group. Each user has his own list of groups he belongs to. Each group is saved as a chunk in ChunKeeper and encrypted using symmetric encryption with the groups master key found in the user’s credentials. The groups chunks are mapped using a key composed of the following tags:

1. The string “groups” - used for labeling this list

2. The group master key. The key is generated randomly, thus we expect each user will have his own different key. This can help both distributing the chunk descriptors more evenly across the chunks DHT and preventing hijacking, spamming and collisions between different users groups list.

3. A sequence number. The sequence numbers main use is to avoid overloading the key value with too many chunks. The other two values are constant for all groups. However, this value distinguishes one group chunk from another. Even though ChunKeeper handles overloading a key with multiple chunks mapped to it, we try to avoid such events lest these chunks will become identifiable by undesired peers.
Whenever a new chunk is inserted to ChunKeeper, stash uses a group to encrypt it. If the group does not exist, Stash will create a new one and insert it to ChunKeeper as well. The group insertion algorithm is illustrated in Figure 4.1.

![Figure 4.1: Insert Group](image)

### 4.4.2.1 Concurrency

One difficult problem emerges whenever lists, such as the groups list, are used with more than one device. The problem is of course keeping the list consistent when multiple instances are involved in inserting and reading data from it. For instance, a user may add a new group both from his smartphone and his desktop simultaneously. In such a case, two groups with the same sequence number may be added. For such cases, ChunKeeper allows the slack of having more than one chunk mapped to the same key. Even though such cases are undesired due to the limitations we inherit from the DHT implementation, we can still handle them in small numbers such as no more than 20 collisions.

### 4.5 Conclusion

In this chapter we have introduced the E-Wolf social network data storage system. ChunKeeper helps us hold the data while Stash provides the interface of encrypting and decrypting it. In the next chapter we will discuss how SocialFS and E-Wolf use stash for organizing the user’s data and sharing it.
Algorithm 4.1 ChunKeeper Store operation

```plaintext
1 function store(key, data):
2     local_storage.store(key, data)
3     replication_factor = calc_replication_factor(age=0)
4
5 // first, ask the nodes we hold their data
6 nodes = select owner_node from storage ordered by rating
7 foreach node in nodes:
8     response = send_STORE_request(node)
9     if response == ACCEPT:
10        HTTPCommunication.send(node, data)
11        replication_factor = replication_factor - 1
12        if replication_factor == 0:
13            return
14
15 // second, ask nodes that already hold our data
16 keys = select key from storage where owner_node = local_node
17 foreach key in keys:
18     chunk_descriptors = findChunk(key)
19     holding_nodes = get_nodes(chunk_descriptors)
20     // remove all the nodes we already asked
21     holding_nodes.removeAll(nodes)
22     foreach node in holding_nodes:
23         response = send_STORE_request(node)
24         if response == ACCEPT:
25             HTTPCommunication.send(node, data)
26             replication_factor = replication_factor - 1
27             if replication_factor == 0:
28                 return
29     // add all the nodes we’ve just now asked to the list
30     nodes.addAll(holding_nodes)
31
32 // third, if we reach here, we still need to find more nodes
33 // to store our data. We have asked all the nodes that we hold
34 // data for and all the nodes that already store other data
35 // belonging to us. Try asking random nodes in the network
36
37 while (replication_factor > 0):
38     key = random
39     node = KeybasedRouting.findNode(key)
40     response = send_STORE_request(node)
41     if response == DECLINE:
42         continue
43     HTTPCommunication.send(node, data)
44     replication_factor = replication_factor - 1
```

Chapter 4 Persistent Storage
5 Online Social Network

5.1 Background

Online social networks (OSN) like Facebook, LinkedIn, and Google+, have become very popular on the Web today. Attracting a broad range of users with different technical skills, these OSNs allow their users to communicate with each other and publish content either publicly or to a selected group of followers. Protecting the users' privacy has never been one of the strong aspects of these centralized social networks since it directly contradicts their financial model of targeted advertising. In this chapter, we describe the final pieces of E-Wolf OSN and demonstrate how all users' data is kept under their control.

5.2 Related work

5.2.1 SafeBook [13]

Safebook consists of a three-tier architecture with a direct mapping of layers to the OSN levels as follows:

1. The user-centered social network layer implementing the social network level of the OSN
2. The P2P substrate implementing the application services level
3. The Internet, representing the communication transport level

5.2.1.1 Architecture

As in E-Wolf, the nodes in Safebook form two types of overlays:

1. A set of matryoshkas, concentric structures in the social network layer providing data storage and communication privacy created around each node.
2. A P2P substrate providing lookup services.

Matryoshkas are concentric rings of nodes built around each member's node in order to provide trusted data storage, profile data retrieval, and communication obfuscation through indirection. Each matryoshka thus protects the node in its
center. The core, which on the social network layer is addressed by its node identifier. The nodes in the matryoshka are connected through radial paths on which messages can be relayed recursively from the outermost shell to the core and vice versa. All paths are based on trust relationships akin to the social network. Thus, each hop connects a pair of nodes belonging to users linked by a trust relationship in real life. The innermost and outermost shells of a matryoshka have a specific role: the innermost shell is composed of direct contacts of the core, and each of them stores the core’s data in an encrypted form. Hence, they are called the mirrors. Every node in the outermost shell acts as a gateway for all data requests addressed to the core, and is thus called an entrypoint.

5.2.1.2 Discussion

Safebook demonstrates an interesting approach to how real life trust can be used to obfuscate communication and preserve privacy. Using the matryoshkas, Safebook is able to hide the identity of both the source and the destination of offline messages. However, it lacks some major features of OSN:

1. Even though protected (encrypted) data sharing is mentioned to be possible, Safebook does not define a safe key sharing mechanism.
2. The replication factor of data is not controlled. No fairness mechanism is implemented either. Thus, entrypoint nodes have no interest in providing a good quality of service.

5.2.2 Confidant [27]

Confidant’s design represents the first decentralized OSN architecture to leverage trustworthy storage servers based on inter-personal relationships. This trust enhances decentralized OSNs by enabling a scalable data-processing framework for third-party applications. Confidant also provides an access-control scheme for weakly-consistent, replicated data that is tailored to the needs of OSN users. In particular, Confidant eliminates write conflicts and allows participants to recover from access-control mistakes by binding access policies to individual data items rather than to groups of items with the same label.

5.2.2.1 Architecture

Confident OSN model is based on three entities:

1. Cloud hosted name servers - Each user runs a name server within a low-cost cloud service such as Google App-Engine. Name servers manage two pieces of state: a list of IP addresses corresponding to online replicas and a logical clock. Name servers also maintain a list of storage servers authorized to act as
5.2 Related work

replicas. A name server’s logical clock is used to assign sequence numbers to items and to help replicas synchronize when they come back online. The value of the logical clock increases monotonically. When a client wants to assign a sequence number to a new item, it requests an increment; the name server responds by adding one to the existing value and returning the new value of the clock. Since only clients under the user’s control should be allowed to advance the clock, increment requests are authenticated.

2. Storage servers - Each Confidant user runs a storage server that contains plain-text copies of all of her objects and access policies. A storage server may also act as a replica for another user if the other user trusts the server’s owner to:

   a) Read all of her objects.
   b) Enforce access policies.
   c) Preserve the integrity of any data-processing scripts run on the server.

3. Clients - Each user may have several different clients, such as a smart phone or a PC, interacting with the other two entities. Clients use the name server for finding the storage servers and the storage servers are used to interact with other clients by updating their data.

Principals in Confidant are defined by a public-key pair, and users are defined by a public-key pair called a root key pair. Users generate their own root keys, and distribute their root public key out of band. A user’s root public key is distributed as a self-signed certificate called a root certificate. The root private key is kept in a secure, offline location. Through her root key-pair, a user also issues certificates for her storage server (storage certificate), name server (name certificate), and any clients under her control (client certificate). These certificates describe the principal’s role and its public key. For each certificate signed by a user’s root key pair, the matching private key is only stored on the component. Users also generate replica certificates with their root key pair for any storage servers controlled by others who are authorized to serve their objects. All certificates are distributed out of band through a service such as Facebook or via email.

Users encode their inter-personal relationships through groups. A group is defined by four pieces

1. A unique user who owns the group
2. A list of users making up the group’s membership
3. An attribute string
4. A secret key used for data encryption.

Clients synchronize the secret key from the group owner storage servers.
5.2.2.2 Discussion

Confidant shows a very practical approach for decentralized OSN. Not all entities in Confidant are decentralized. However, the few that do remain semi-centralized, the name servers, are only providing a thin layer of communication infrastructure. Yet, its permissiveness storage handling comes to a great drawback. Not only storage servers hold plaintext data of users who do not control them, but also they are hand picked. Having several storage servers replicating your data makes it hard to identify the source of a data leak should it ever occur. Moreover, selecting your storage servers intelligently requires an above average computer security understanding making the laymen prone to simple phishing attacks by convincing her to use a certain storage server.

In addition, secret keys sharing relies on storage servers enforcing correctly access control permissions. Should a group’s secret key become compromised either by storage server contents stolen or invalid handling of ACL permissions, an adversary may have access to the group’s future messages.

5.2.3 Contrail [35]

Contrail is a communication platform that enables efficient, decentralized social networks on smartphones. At the heart of Contrail is a simple cloud-based messaging layer that enables basic connectivity between smartphones, allowing them to efficiently and securely exchange encrypted data with other devices.

5.2.3.1 Architecture

Contrail consists of a client-side module that executes on each device, and a messaging layer that resides in the cloud. Each client-side module periodically initiates a TCP connection to the cloud-based messaging layer via 3G (or a WiFi hotspot). In simple terms, a message sent by one device to another is first uploaded to the cloud via one device-to-cloud connection, and subsequently pulled by the recipient device via another such connection.

When devices connect to the cloud, they interact with one of the application servers; we call this the proxy for the device. If a device uploads a message meant for an offline recipient, its proxy stores the message in the storage tier. When the recipient device comes online, its proxy checks the storage tier for any messages meant for it and transfers them. Privacy is ensured via device-to-device encryption: the cloud sees only encrypted payloads. Contrail strategy for encrypted communication is not novel; it uses simple off-the-shelf techniques. It uses public key encryption to exchange symmetric keys between devices, which are then used for encrypting all messages.
5.2 Related work

5.2.3.2 Discussion

Contrail represents a solution for the smartphones three major problems:

1. Online availability - Smartphones tend to connect and disconnect form the network frequently, whether it is an elevator or simply a bad reception area, the device will go offline.

2. Battery capacity - Since battery capacity is an issue, you cannot route chunks of data too frequently. For example, participating in a DHT requires a lot of ping packets answered and a lot of queries answered. Both tend to spread over time, making the device unable to switch the network to power save mode.

3. NAT - Due to large number of mobile devices, CGNAT (Carrier-grade Network Address Translation) is not uncommon. These network asymmetry makes it impossible to initiate a connection between two devices without a helper intermediate, such as a server in the cloud.

Although Contrail fits very well for smartphones, it is far from being privacy friendly. Clients periodically query the cloud for new messages, allowing the cloud to track their position using Geo IP services. Moreover, all data is saved in the cloud. Should a symmetric key between two clients ever got compromised, all past communication between those clients will be readable to the cloud.

5.2.4 PeerSoN [8]

PeerSoN aims at keeping the features of OSNs but overcoming two limitations: privacy issues and the requirement of Internet connectivity for all transactions. To address the privacy problem, PeerSoN uses encryption and access control coupled with a peer-to-peer approach to replace the centralized authority of classical OSNs. These two measures prevent privacy violation attempts by users, providers, or advertisers. Extending the decentralized approach, PeerSoN also enables direct exchange of data between devices to allow for opportunistic, delay-tolerant networking. This includes making use of ubiquitous storage to enable local services.

5.2.4.1 Architecture

PeerSoN defines the following protocols:

1. Login - A login to the system is the announcement that a certain user is now online along with the meta-data necessary to connect to this user and a list of files that the peer stores. This announcement is sent to OpenDHT in the format of a key-value pair.

2. Getting File - Whenever Alice wishes to download a file from Bob, she first needs to download Bob’s index file from the OpenDHT. In the index file Alice
can find out all of Bob’s updates. All files, including index files are protected with appropriate access control mechanisms in the design of PeerSoN.

3. Asynchronous Messages - The sender stores messages within the DHT until the receiver logs in again and pulls all the messages. He then sends a delete request to the DHT in order to delete those messages.

4. Synchronous Messages - A handshake protocol is implemented for peer to peer file transfer.

5.2.4.2 Discussion

As in E-Wolf, PeerSoN, heavily relies on the DHT infrastructure for communication between peers. However, the entire project lacks of other components. PeerSoN does not describe a persistent storage system such as ChunKeeper and its integration. Their asynchronous messages model requires a DHT delete message, which can be easily used for malicious deletions.

5.2.5 Diaspora

Diaspora network is composed of Pods. Pods participate in the Diaspora Peer to Peer network and implement the distributed social network protocols. Each Pod may host several users accounts. When joining Diaspora, users need to choose the Pod where their account will be hosted. Pods act as Web servers providing access to users accounts via browser based clients and enforcing ACLs.

5.2.5.1 Discussion

Diaspora innovation is separation between the infrastructure (storage, bandwidth, uptime, etc) and the logical service of participating a large social network. Users of different Pods interact with each other and choose Pods based on their judgment. Pods may provide their services for monthly fee or may inject advertisements to their users’ wall. Some may even donate their services. While Diaspora architecture partially solves the issue of one quality target, it still requires an infrastructure of a highly available servers. Users still need to trust their Pod not to violate the agreement between them as all their data is shared with their Pod. Moreover, users account availability is depended on their Pod’s availability. Whenever the Pod suffers a downtime or storage failure, all the hosted accounts become unavailable and data may be lost.

5.2.5.2 Related work conclusion

In these related work we have seen several examples of OSNs and their place on the centralized-decentralized spectrum. Each work has located itself on a different place,
some were more centralized, such as Contrail that emphasized usability. Others were more decentralized such as Safebook and PeerSon that use the DHT for data retrieval and nodes synchronization. And there are those who try to take the advantages from both worlds, such as Confidant that uses a cloud service name servers to locate other entities. E-wolf is located at the far decentralized edge of this spectrum, were it requires no centralized service whatsoever and all node communication is done using the DHT.

5.3 SocialFS

As ChunKeeper provides the persistent data storage and Stash encrypts it, SocialFS organizes the chunks in a hierarchal tree like form. All data in SocialFS is encrypted with different symmetric keys to ensure control over the data viewers while balancing performance issues caused by encrypting data to each viewer separately. SocialFS most important assumption is that all data a user generates is immutable. Thus, files in SocialFS cannot be modified easily. However, appending new data to a file’s context, such as commenting on a post, is simple and supports multiple accesses simultaneously.

5.3.1 Architecture

SocialFS’s basic chunk of data is called a “File”. Each file holds several properties allowing it to be located and additional files can be associated with it:

1. The file name - an arbitrary name given to the file.
2. The file content - arbitrary data the user has set.
3. File key and parent file key - randomized keys used for identifying the hierarchichal connection between two nodes.
4. A list of associated files keys - these keys are slots for other files to be associated with the file in hand. Once all slots but the last one are occupied, a special file is generated without a name or content. This file simply allows more files to be associated with the parent file by extending the number of available slots.
5. Signature - the file is signed by its owner’s asymmetric signature key for authentication.

Each file is mapped to a single chunk under the file’s random key. Stash encrypts it using the selected group. Each user has a single root file which is the ancestor of all his files. The root file is encrypted using the general “friends” key shared by all the user’s friends in E-Wolf and mapped in Stash under the root key. The distribution of keys is discussed later in the E-Wolf chapter.

Figure 5.1 illustrates files relationship in SocialFS

\[1\]https://code.google.com/p/social-fs/
5.3.1.1 Meta-data

SocialFS has two types of meta-data elements:

1. The user profile - It contains all the public information a user shares with the entire network: public signature and encryption keys and the root key. Users profiles are inserted to the “profiles DHT” periodically whenever the user is online. The profiles are mapped under the UserID, which is a digest of the user’s public signature key, allowing users to find each other’s profiles. Users may choose to map their profile under several keys such as their names for participating in the general user search.

2. The user credentials - A file saved encrypted in Stash as discussed in the previous chapter. It holds the user’s Stash group master key as well as the user’s private signature and encryption keys. The file also includes the user’s profile where the matching public keys are. Note that this file is never shared and is encrypted by the user’s own password.
5.3 SocialFS

5.3.1.2 Operations

SocialFS has four simple operations:

1. Login - Find the user’s credentials and decrypts them.

2. Profile search - Search for other users’ profiles, either by explicit UserID or by name.

3. Create new file - Wrap arbitrary data in a file element and insert it to Stash. The user must specify the file’s group and parent file. The file is encrypted using the selected group key and inserted to the next available slot of associated file keys of the parent.

4. Retrieve a file - Traverse files the user has access to, either his own or belonging to other users with whom he has a shared key with. Once the file is found, it is decrypted using the appropriate key.

5.3.2 Consistency

In SocialFS, users may only associate files they create with their own files. File association is enforced using signatures. For example, Alice tries to associate her file with Bob’s root file by inserting a file of her own with the parent key property set to one of Bob’s root file associated keys. Whenever Carlo traverse Bob’s files and encounters Alice’s file, she can ignore it since Alice’s file is not signed using Bob’s signature. The only way for Alice to sign her files with Bob’s signature is to find out Bob’s secret signature key in Bob’s credentials.

In Figure 5.2, when Carlo tries to traverse Bob’s file with the key “aaa111” he will find both Alice’s and Bob’s files. Since Alice’s file does not have Bob’s signature on it, Carlo will ignore it and only retrieve Bob’s file content and associated files.

This property allows us to assume that there is only one user with writing privileges to his own files tree. However, this user may be using several clients that may add files simultaneously. SocialFS allows the slack of adding more than one file with the same key. However, when traversing the file system, the order of these files are not guaranteed. Even though this may be a significant drawback for file storage systems, the main usage of file association in E-Wolf is for commenting. It is permissible for two or more comments that were added simultaneously to appear in different order when different users try to view them. Note that the consistency of the entire tree is not broken, but only the order is not guaranteed.
5.4 E-Wolf

E-Wolf is the last component of the E-Wolf distributed social network. It provides a simple interface for adding posts and albums as well as secure key sharing service for allowing users to communicate.

5.4.1 Architecture

E-Wolf provides three types of messages a user can send to another:

1. Text message - A simple message containing an arbitrary text.

2. Poke message - A message containing a secret key which another user wishes to share. Should the user chooses to accept the key, E-Wolf will save it in Stash’s groups list.

3. Comment message - An arbitrary data, can be text, picture or video, with a file’s key it is associated with. Should the user chooses to accept the com-

\[\text{https://code.google.com/p/ewolf/}\]
ment, E-Wolf will add it to SocialFS associated file. The comment content is encrypted using the associated file’s group key, thus proving that the commenting user knows it. Note that the comment is signed by both the user who wrote the comment and the owner of the associated file, hence, forged comments are prevented.

All messages are encrypted using the addressee user’s public encryption key which can be found in his profile and signed using the addresser’s private signature key. All messages contain within themselves the addresser’s profile key allowing the addressee to verify the message signature. E-Wolf uses the “messages DHT” to send messages. Each user has a messages feed key which is the digest of his UserID and “messages”. When Alice wishes to send Bob a message, she simply inserts the encrypted message to the messages DHT mapped under Bob’s messages feed key. All Bob has to do is pull all messages from his feed periodically. Messages in the messages DHT have short lifetime. They are deleted after a few days and reinserted again since they are not saved persistently anywhere.

E-Wolf organizes the user’s peers in wolf packs. Each wolf pack has a Stash group associated with it and a list of members. The wolf pack and the members are mapped to files in SocialFS. The members files are all associated with the wolf pack file and the wolf pack file is associated with the root file and named “wolf-pack” as illustrated in Figure 5.3. Adding members to a wolf pack is easy. Simply add a new file with the member’s UserID and associate it with the wolf pack file. However, deleting a member from a wolf pack is much harder. A new wolf pack file is created and all the members are copied to the new file without the removed member of course. A poke message is sent to all the pack members, updating them with the new group secret key. This procedure has significant performance costs. Nevertheless, it is mandatory since SocialFS does not allow us to modify files. The old wolf pack will be kept in order to allow old posts decryption. Note that the deleted user can still view old posts up to the time he was deleted.

Aside from messages, a user may upload a post. A post may be any combination of text, video, and pictures. Each post is encrypted using the selected wolf pack’s group key. When other members of the wolf pack view the user’s wall, they will be able to decrypt the post and send comment messages using the post file key. All posts are files in SocialFS and are associated with the Wall file which is associated with the root file and named “wall”.

Another important file of the user’s social space is the “albums” file. The albums file contains “album” files associated with it. Each album file has picture files associated with it. As with posts, every album is associated with a wolf pack and all its pictures are encrypted using the wolf pack secret key.
5.5 Conclusion

In this chapter, we have seen how all the parts are glued together. SocialFS defines a tree like hierarchy for every user and E-Wolf uses it to organize the user’s data in a few folders. The wall folder containing the user’s posts, the “wolf-pack” folder containing the users groups and the “albums” folder contains the user’s pictures. As we can see, E-Wolf provides the basic social network requirements and can be easily extended to include future data types.
6 Kaleidoscope: Adding Colors to Kademlia[14]

6.1 Abstract

Kademlia is considered to be one of the most effective key based routing protocols. It is nowadays implemented in many file sharing peer-to-peer networks such as BitTorrent, KAD, and Gnutella.

This chapter introduces Kaleidoscope, a novel routing/caching scheme designed to significantly reduce the cost of lookup operations in Kademlia by using a color-based distributed cache. Moreover, Kaleidoscope greatly improves load balancing among the nodes and reduces the well documented hot spots problem. The chapter also includes an extensive performance study demonstrating the benefits of Kaleidoscope.

6.2 Introduction

Distributed Hash Tables (DHT), and in particular, their key-based routing (KBR) schemes, are at the heart of most peer-to-peer (P2P) systems. Consequently, a plethora of papers and ideas on how to implement DHTs has been published, e.g., [5, 29]. DHTs tend to differ from each other in the routing scheme they employ, as well as the space and message overhead they incur for maintaining their overlay. As the DHT employed by many popular file sharing applications, Kademlia has become one of the most widely used DHTs in practice [30]. This is largely due to its proven robustness to churn and relatively fast lookup time, enabled by its unique partially parallel lookup mechanism, large routing tables, and adaptive learning routing process.

One of the well documented problems of all DHTs is hot spots [21, 28]. That is, a node that stores a key/value for some content that is very popular becomes overloaded to the point that it can no longer serve any request. A partial solution to this problem is to employ caching. As DHT routing usually takes multiple hops from source to destination, the nodes on the way can cache the results of requests for popular content. This way, they can send the cached copy back to the requester.

However, caching is of limited help in overcoming the hot spot problem in Kademlia, since routes in Kademlia constantly evolve. To tackle this problem, we introduce
a novel caching and augmented routing mechanism for Kademlia, called Kaleidoscope. Kaleidoscope eases the Kademlia hot spot problem by employing both a secondary hashing technique that we call colors and an overload protection mechanism. Combined, these methods achieve this goal while employing relatively small caches. Kaleidoscope also reduces the overall traffic in the system and outperforms other known Kademlia caching schemes.

Specifically, like many DHTs, in Kademlia each node and each item are assigned a hashed key and roughly speaking, each node is responsible for the items whose hashed keys are closest to its own hashed key. Also, Kademlia employs a suffix based routing, yet uses a XOR metric for that. Moreover, Kademlia employs relatively large routing buckets, which are constantly being updated as each node learns about other nodes in the system. Lookups in Kademlia are performed using a partially parallel process, in which the searcher tries to iteratively locate the closest node to the search key.

In Kaleidoscope, we assign each node and each item an additional color, obtained by hashing their Kademlia key over a small range of colors. Loosely speaking, we augment the search process of Kademlia to prefer contacting nodes whose color is the same as the item being sought (if exists). This way, we increase the likelihood of hitting a node that already has the corresponding value cached.

A find value operation in Kaleidoscope is performed in two stages: First, we forward the request to a correctly colored node, i.e., a node whose key matches the color of the value we seek. We argue in this chapter that such a forwarding can potentially be done with low latency and short timeouts. Once we reach the correctly colored node, if the value is in its cache, then the result is sent back to the requester. Otherwise, that node will begin a parallel lookup process in order to quickly find the value, cache it and send it back to the requester.

Note that such forwarding can also potentially increase the overall message overhead and latency, since it adds messages to the routing process. Hence, the difficulty here is in realizing this seemingly simple idea while ensuring that the search still always advances toward the requested key and avoiding increasing the load on nodes that hold popular values, as we elaborate below.

Next, we present a formal analysis of the expected behavior of Kaleidoscope followed by an extensive experimental performance evaluation. In the experiments, we have observed that the formal analysis is fairly indicative of the actually measured results and have used them to tune the parameters of Kaleidoscope.

Further, we have implemented and tested KadCache, a distributed caching suggestion of the Kademlia authors [30]. This caching suggestion was intended to reduce hot spots but was never implemented and tested before. We have also implemented the local cache suggested in [15], a.k.a. Local. Last, we present a performance comparison of Kaleidoscope with KadCache and Local, and show that Kaleidoscope outperforms both of them in terms of both overall system load and in terms of hot spot handling.
6.3 Related Work

Several works have investigated how to use caching to reduce the lookup cost in DHTs. For example, in [15] it is suggested to add to Kademlia a local cache named Fast Table. This table stores the results of previous lookups the node has performed. When a node receives a find value request, it first checks its Fast Table to see if it contains cached results for it. While this approach yields a significant 10% reduction in average lookup length, it is not sufficient to deal with the more serious hot spot problem, and does little to balance the bandwidth load in the network.

Another important caching suggestion appears in the original Kademlia paper [30]. In this suggestion, every time a node performs a find value operation, it sends a store value request to the last node it contacted that did not have the value. As far as we are aware of, this suggestion was not evaluated before. While the authors of [30] suggested that it will alleviate the hot spots problem, we have found that its ability to do so is limited.

On a more theoretical note, an important observation about using caching to cool hot spots is made by [6]. That work has explicitly identified the hot spot problem for Kademlia and similar DHTs when faced with an uneven distribution of value popularity. They modeled the problem and proved that caching alone is unable to deal with the hot spot problem. They also suggested that in order to cool hot spots the caching scheme and the routing algorithms should be aware of each other. Additionally, they designed a routing/caching scheme that significantly improved the load distribution, but did not reduce the overall message cost.

The work of [9] deals with load balancing on KAD networks (that are based on Kademlia for file sharing). In such networks, searches are stopped once they collect enough download sources. These networks experience hot spots due to popular search queries. In this suggestion, load balancing is achieved by scattering references for popular content in larger and larger cycles around the closest node. Although this suggestion is not entirely a caching suggestion, it is similar to the KadCache suggestion of the original Kademlia authors. Kaleidoscope can augment this suggestion by offering attractive locations to store download sources of popular files. In particular, the nodes to which popular sources can be scattered to could be of the same color as the value being stored.

It is also worth mentioning that [37] managed to reduce the lookup costs in Gnutella using a combined routing/caching approach. This is done by having the nodes and
Chapter 6 Kaleidoscope: Adding Colors to Kademlia[14]

the keys being spliced into a small amount of distinct groups that are similar to our colors. When performing lookups, instead of flooding the lookup to all the nodes at certain radius it was suggested to propagate the lookup only to the nodes of the appropriate group. This change in routing was combined with a slight change to the default caching scheme, by which nodes only cached items of their own group id. Although this suggestion resulted in a significant improvement of Gnutella, by the time it was published more advanced DHTs like Chord [34] and Kademlia already started replacing Gnutella and offered significantly more efficient lookups than the improved Gnutella.

Other non cache related methods that were offered in order to reduce Kademlia’s lookup costs include techniques to fill k-buckets with nodes of geographical proximity [10, 23] and a recursive lookup scheme [18]. On a different note, other DHTs like OneHop [16], Kelips [17] and Tulip [3] achieve O(1) lookups at the cost of background traffic overheads. The latter also considers proximity in its routing.

6.4 A Brief Overview Of Kademlia

Kademlia is described in [30]. Here we only give a brief overview of its structure and main properties as a background to this chapter. In Kademlia, each node and each object are assigned a 160-bit key using a hashing function (such as SHA-1). The notion of distance is defined in Kademlia using the XOR metric and objects are stored in the nodes whose id is closest to theirs according to this metric.

In Kademlia’s overlay, each node maintains a bucket of up to k nodes (or k-bucket) for each of the 160 bits of its key. The k-bucket corresponding to the i-th bit of node p stores the k closest nodes that p is aware of whose distance from p is between \(2^i\) and \(2^{i+1}\). This overlay may constantly evolve as p learns about additional nodes.

Each k-bucket is kept sorted by the least-recently seen order. Each time p receives a message from q that corresponds to a given k-bucket of p, if q does not exist there and the bucket is not full, it is added to the bucket as the most-recently seen node. If q already exists in the bucket, it is simply updated to be the most-recently seen node there. Finally, if the bucket is full and q is not there, then the least-recently seen node r in the bucket is pinged; if r does not answer, then r is deleted from the k-bucket and q is inserted as the most-recently seen node. Otherwise, r is updated as being the most-recently seen node.

The routing process of Kademlia proceeds as follows: When a node p needs to find the node (or value) corresponding to object o with key d, p begins the following iterative partially parallel lookup process. p creates a list of the k closest nodes to d that p is aware of (possibly including itself); call this list the k-candidate list for d. It is possible that this list may contain fewer than k nodes. Then, on each iteration, p picks the first \(\alpha\) (a parameter greater than 0) in the k-candidates list that has not queried yet and sends each of them in parallel an asynchronous lookup query
for d. In response, each of these nodes r returns to p the k closest nodes to d that r is aware of. When p receives the reply, it updates its k-candidates list to hold the k closest nodes to d that p knows about following the receipt of this reply. Also, after each such reply, or a timeout on a given recipient, p sends a query to the first node in its k-candidates list that it has not contacted yet (having up to \(\alpha\) outstanding queries at any given time). This process ends when p has finished contacting all nodes in its k-candidates list. At this point, the looked up nodes are declared as the k closest nodes to d that p is aware of (essentially, the remaining nodes in p’s k-candidates list).

Similar to many DHTs, it has been proved in [30] that the lookup time of Kademlia is at most logarithmic. In contrast with some other well known DHTs, such as Chord [34], the Kademlia overlay constantly evolves, yet need not be updated immediately following joining or departure of nodes. Also, each entry has k nodes, giving the routing protocol some freedom in choosing its preferred route. Hence, two consecutive lookup requests for the same item may follow different routes even if there is no churn in the network.

### 6.5 Kaleidoscope

#### 6.5.1 Colors

In Kaleidoscope, we assign to each node and each (key,value) pair, in addition to their Kademlia key, another secondary key called color. The color is generated by hashing the Kademlia key. Unlike the Kademlia Key, many nodes/objects are likely to get the same color. In particular, we wish the colors’ domain to be of a similar magnitude as the Kademlia bucket.

Additionally, each node holds a small local cache for storing values. We define the color of the cache to be the color of its node. When a node p obtains a (key,value) pair, p stores this (key,value) in its cache if and only if both are of the same color. When the local cache becomes full, we use the standard least recently used policy for evacuation.

#### 6.5.2 The Routing Scheme

We have modified the original Kademlia routing scheme as follows (see Algorithm 6.1 and Algorithm 6.2 for pseudo code and Figure 6.1 for an example): First, we have added a request forwarding phase, in which nodes forward the find value request until they encounter a node of matching color.

Specifically, when a node p initiates a find value request, as in Kademlia, it first computes the k nodes it is aware of that are closest to the desired object key according to the XOR metric – this is the k-candidates list mentioned above. Yet,
Algorithm 6.1 The Forwarding Scheme

1 function ForwardFindValue(findValueRequest v):
2    if myKey.Color == v.Color:
3        // Stop 1: correct color
4            X = doFindValue(v)
5            sendResultBackwards(X)
6            return
7
8    v.kCandidates = getKClosest(v.key, v.kCandidates, myKBuckets)
9    cn = getClosestNode(v.key, v.kCandidates)
10   if distance(v.key, myKey) <= distance(v.key, cn.key):
11      // Stop 2: cannot advance in XOR metric
12        X = doFindValue(v)
13        sendResultBackwards(X)
14        return
15
16   // Try and forward result
17   nextHop = kCandidates.getByColor(v.Color)
18   Success = sendForwardFindValue(nextHop, v)
19   if not Success:
20      // Forward may fail if node is congested
21        MarkBusy(nextHop)
22        sendResultBackwards(X)
23        return
24
unlike Kademlia, in the request forwarding phase of Kaleidoscope, the request is then forwarded, along with the k-candidates list to one of its members q who is closer than p to the target object key (in the XOR metric). This node q is chosen as the closest one in the k-candidates list who has the same color as the target object key, if exists, or simply the closest if none of the nodes in the k-candidates list has the same color as the object key (Line 17 in Algorithm 6.1). Each node remembers the lookup id and sender in order to route the lookup back. This information is erased once the lookup is routed back or following a timeout.

When q receives the forwarded lookup request, it computes its k-candidates list by merging its own Kademlia based knowledge with the list of nodes incorporated in the forwarded request (Line 8 in Algorithm 6.1). If q’s color is the same as the key of the request, then q will perform the iterative find value process by itself. Otherwise q continues the forwarding phase. That is, q forwards the request either to a node in its k-candidates list who has the correct color, or to the closest node to the requested key that is in its k-candidates list.

Hence, the forwarding phase ends when the message reaches either a node of the correct color (Line 2 in Algorithm 6.1) or a node who is closer to the request’s key than any other member of its k-candidates list (Line 10 in Algorithm 6.1). In the extreme, this forwarding phase can be empty in case the originator already cannot find any closer node to the object key than itself, or if it is of the correct color. Yet,
6.5 Kaleidoscope

**Algorithm 6.2 The Find Value Scheme**

```plaintext
1 function DoFindValue(findValueRequest v):
2     X = cache.search(v)
3     if X.value != null:
4         return X.value
5     while hasUnQueried(v.kCandidates):
6         node = takeUnqueriedSortedByColor(v.kCandidates)
7         X = queryNode(node)
8         if X.value != null:
9             return X.value
10        MarkNodeAsQueried(node)
11        v.kCandidates = KClosest(v.kCandidates, X.nodes)
12     return X.value
```

we never forward a request to a node that is not closer to the requested key even if it is of the correct color. This is to ensure that the forwarding phase is bounded in the worst case by $O(\log n)$ steps.

Once the forwarding phase ends, the node at which it ended starts the find value phase. As mentioned before, this phase is almost identical to the one in Kademlia, except that in each step, if one of the nodes in the k-candidates list has the same color as the requested key, and this node has not been contacted before, then this node is contacted (Line 6 in Algorithm 6.2). Every node that receives a find value request, checks if the value is stored in its cache. If yes, then the value is returned from the cache. Otherwise, the find value phase is continued. Once the iterative phase ends, the request is forwarded back to the originating user.

For performance reasons, when node q receives a forwarding request from p, q will send back to p an ACK, signaling that the message was forwarded successfully. If q also has the result in its cache, q will send the requested value instead of the ACK. q can also send back a partial result if it is too busy to accept the forwarding request, as explained in the overload protection section below.

### 6.5.3 Overload Protection

A well documented feature of Kademlia is over representation of long lived nodes. In Kademlia, long lived nodes receive more incoming bandwidth because they tend to appear more often in routing tables of other nodes. This is because when joining Kademlia, each node picks several bootstrap nodes uniformly at random, and alive nodes are never thrown out of a Kademlia bucket. This problem is escalated when employing a hybrid mechanism like we suggested. The reason for that is that now, the long lived nodes with high incoming degree are significantly more likely to receive a forwarded lookup.

In Kaleidoscope, these nodes do not need to just answer the request, but to actually perform the rest of the iterative lookup. Hence, without an additional mechanism,
these nodes may become overloaded, and slow down the system. In order to address this problem, we introduce a simple load sharing protocol that we name Overload Protection.

The basic idea of this protocol is that a node can reject forwarding attempts if it is too busy. There are many possible ways of defining when a node is busy. In our prototype, each node has a very short queue of pending lookups to execute. Once this queue fills, the node will not accept additional forwarding attempts. The motivation of a short queue is to keep the lookup execution delay low.

When rejecting a forward request, the node merges the relevant k-bucket with the forward request’s k-candidates list and returns the result as a NACK. The lookup will then propagate back to the originating node who will perform an iterative lookup. The reason for this is that a rejected forwarding attempt may imply a momentary congestion in the network and a higher risk for timeouts. The resources that were already invested in the lookup are not lost as the k-candidates list contains a partial result. The initiator node uses this list to continue the lookup from the point it stopped. In addition, when a node receives a NACK (or does not receive a reply at all), the sender is marked as busy. Since busy nodes are less useful in k-buckets, this replaces busy nodes with fresh nodes without even sending a ping.

An interesting feature of this overload protection protocol is that it allows overloaded nodes to reduce their representation in routing tables and therefore reduce some of their incoming traffic. Experiments shows that overloading do not always happen

Figure 6.1: A routing example for Kaleidoscope
due to excessive numbers of incoming forward request. Some of the nodes store very popular values and therefore take longer to perform lookups and are more likely to send a NACK.

However, when nodes that store popular values reduce their representation in routing tables, the overall performance of the system decreases a bit. Yet, these nodes manage to reduce their incoming traffic and as lookups for their popular values become slightly longer, it increases the chances of hitting a cache on the way.

### 6.5.4 Short Forwarding Phase

Recently, the authors of R-Kademlia [18] showed that under the churn rates experienced in real KAD networks, lookup bandwidth can be reduced using a recursive forwarding scheme. The recursive forwarding was then combined with proximity neighbor selection and routing techniques, in order to achieve comparable latency to iterative routing.

Yet, applying a forwarding scheme to Kademlia has its drawbacks. First, it significantly reduces the number of nodes encountered. Since Kademlia lookup messages serve as an additional k-bucket refresh mechanism, recursive forwarding results in additional k-bucket refresh overheads. This is significant since recursive forwarding has no parallelism and thus requires a more intensive table refresh policy.

Second, the length of a Kademlia lookup is $O(\log(n))$ and therefore the stability of a forwarding path depends not only on churn rates, but also on the size of the network. Notice that nodes may fail a forwarded lookup even if there is no churn at all due to overload that causes them to be too busy to answer.

We also observe that proximity neighbor selection techniques are significantly more likely to populate the first k-buckets (corresponding to longer XOR distances) with close by nodes, as there are many potential candidates to populate them. Other k-buckets have a much smaller selection of potential candidates and a proximity protocol is less likely to find close by nodes for them. We therefore chose to use a hybrid approach in Kaleidoscope that combines an initial short forwarding phase followed by a parallel iterative lookup phase as it seems to combine the best of both worlds.

We begin with forwarding the message to a correctly colored node. This forward is of limited length and can therefore be done efficiently with proximity neighbor selection methods. At that point, if there is a cache miss, we switch to an iterative lookup that is the quickest method to complete the lookup. It is important to finish quickly since a color cache miss indicates that we are likely not to hit caches at all during the lookup. If there is a short forwarding timeout or a NACK, the lookup is rolled back and performed iteratively to minimize the latency impact. In this case, we also use the already calculated partial lookup result instead of starting from scratch.
6.5.5 Discussion

Routing in Kaleidoscope can reduce the bandwidth cost of lookups by employing a hybrid forward/iterative routing scheme. At first, Kaleidoscope performs a short forwarding step that ends at a node that is likely to have a cached copy of the value. The forwarding step is short and does not depend on the size of the network. It is also part of the overload protection protocol, which helps dealing with hot spots.

Nodes in Kaleidoscope usually only perform requests for objects of their own color. They are therefore more likely to contain cached copies of such values. Kaleidoscope also enhances the iterative phase by preferring nodes of the request color when possible. In both phases, Kaleidoscope always advances the lookup to its goal according to the Kademlia XOR metric. Hence, lookups in Kaleidoscope attain the same \( O(\log n) \) hop count bound as Kademlia. In addition, when congestion or churn occur, forwarded lookups are rolled back and the initiator is able to continue the lookup using the partial result already calculated.

Notice that we cache replies only in nodes with the same color as the request object key. As discussed in Section VI below, we have found that this greatly enhances the ratio between the cache size and the cache hit rate compared to always caching the result. Yet, this optimization can be eliminated when unlimited caches are allowed.

6.6 Analysis

Throughout the probabilistic analysis below, we make several simplifying assumptions as detailed below. Yet, as demonstrated by our performance evaluation in Section VI, the actual results follow closely our analysis, suggesting that the overall analysis is indicative despite these assumptions.

6.6.1 Expected Performance

6.6.1.1 Simplification assumptions:

Assumption 1: The forwarding path ends at a correctly colored cache. That is, we ignore the case where the forwarding scheme is routed all the way to the value. Such cases are edge cases that either happen with very low probability (very long forwarding path) or happen when a node requests a value that is very close to its hashed identity, which also happens with low probability in the system.

Assumption 2: Forwarding paths can end at any node (of correct color) with the same probability. Note that when this is not the case, the performance of the cache is significantly better since part of the caches receive more traffic than their fair share. Nodes using these caches experience higher hit rate, and the benefit applies to a higher percentage of the nodes improving the overall system performance.
6.6 Analysis

Assumption 3: All nodes request approximately the same number of values at an arbitrary common distribution.

Assumption 4: Keys are colored uniformly at random, which is reasonable given that keys are generated through a hashing function and colors are obtained through a secondary hashing on these keys.

Assumption 5: The aggregated popularity of all the keys from each color is exactly the same.

Assumption 6: The network size is very big and the caches are warmed up.

6.6.1.2 Cache hit rate:

Denote by $P_d$ the popularity of a key $d$, i.e., the percentage of nodes that have issued in the past a find value request for $d$. The probability that the infinite cache of a node $q$ does not contain a pair $(d, *)$ (with any value $*$) is the same as the probability that $q$ never inserted $(d, *)$ to its cache in the past. That is, none of the nodes in the system has previously forwarded a find value request for the key $d$ to $q$.

Recall that under our assumptions, all forwarding paths end at a correctly colored node and each node is as likely to be the end point of a forwarding path. The forwarding takes $n$ requests from $n$ nodes and delivers them to a subset of $\frac{n}{C}$ nodes. Hence, each node is expected to be the endpoint on average for $C$ nodes should they request a key of its color.

$$P_{miss} = P((d, *) \notin q.\text{cache}) = \begin{cases} 1 & \text{if color}(d) \neq \text{color}(q) \\ (1 - P_d)^C & \text{otherwise} \end{cases}$$

Insert formula

For example, if we consider a value of popularity 1% ($P_d = 0.01$), then an arbitrary local cache will have a hit with probability of only 1%. However, the hit probability of a 17 colors cache is approximately 15.7%. Yet, evaluating the hit-rate is not enough, since we do not always find a node of the correct color right away.

6.6.1.3 The length of the forwarding path:

We now estimate the number of forwarding steps in the request forwarding phase as a function of $k$, the size of the Kademlia $k$-bucket (and $k$-candidates list) and the number of colors $C$. For the purpose of this analysis, we further assume that the $k$-candidates list is full and that in each forwarding step we get $k$ new candidates.

Assumption 5, ensures us that the probability to request a value of a specific color is the same for all color. Assumption 4, ensures us that all the nodes in the system are colored uniformly at random. If we combine both we can conclude that the
probability of any given node that issues a find value 1 request to be of the correct color is $\frac{1}{c}$. In other words, the probability of not being of the correct 1 color is $1 - \frac{1}{c}$. Thus, for a given k-candidates list, the probability that none of its members is of the correct color 1 is $(1 - \frac{1}{c})^k$. Therefore, the expected forwarding path length can be expressed as a geometric random variable as follows:

$$P(L_f = 1) = (1 - \frac{1}{c})(1 - (1 - \frac{1}{c})^k)$$

$$P(L_f = 2) = (1 - \frac{1}{c})(1 - \frac{1}{c})^k(1 - (1 - \frac{1}{c})^k)$$

$$P(L_f = i) = (1 - \frac{1}{c})^{k+1}P(L_f = i - 1)$$

$$\mathcal{L}_f = \sum_{i=1}^{m} iP(L_f = i)$$

where $m$ is the maximal forwarding path length. Since it is hard to determine $m$ analytically, we will bound the path length using the asymptotic formula for a geometric variable (it is reasonable under Assumption 6). $\mathcal{L}_f$ can be bounded as follows:

$$\mathcal{L}_f = \sum_{i=1}^{m} iP(L_f = i) \leq \sum_{i=1}^{\infty} iP(L_f = i) = (1 - \frac{1}{c}) \frac{1}{(1-(1-\frac{1}{c})^k)}$$

For example, if we choose $k = 7$ and $c = 7$, we are expecting the forwarding path to be about 1.3 hops. If, however, we choose $k = 7$ and $c = 17$, we expect the forwarding path to be no longer than 2.72 hops on average.

### 6.6.1.4 Expected message cost of the forwarding path:

In our protocol, when a node receives a forwarding request, it has 3 possible responses: In case of a cache hit, it responds with the requested value; in case of a cache miss it can either respond with a partial result if the node is busy or respond with an ACK if it accepted the forwarding. Under the simplifying assumptions of the formal analysis model, nodes always accept forwarding requests and therefore in case of a cache miss, the node responds with two different messages. The first message is an ACK signalling the acceptance of the forwarding, and the second message is the value requested. In summary, each forwarding step generates 3 messages (the forward request, the ACK, and the final response), except possibly for the last one. In the last step, a hit generates only two messages (the forward request and the response) while a miss generates three. Hence, the expected cost of the last step is $2 + P_{\text{miss}}$. Therefore, for a forwarding path of length $L_f$ the average message cost is as following: $\text{ForwardingCost}(L_f) = 3L_f - 1 + P_{\text{miss}}$

### 6.6.1.5 Expected number of lookups during the iterative find value process:

Let us first consider a find value operation with concurrency factor $\alpha = 1$, which is easier to analyze. In this case, the find value initiator sends a single find value
request to a node from the correct bucket that either has the same color as the requested key, or is the closest node to the key. The initiator then waits for the results, and only sends an outgoing message once a reply arrives.

We call each message sent and corresponding response a step and assume that on each step of the find value process, the node performing the lookup obtains a complete set of k new candidate nodes. This is true when the node being contacted has the appropriate k-bucket full, as we assume in our model.

This implies that each node that is being contacted provides k other nodes that are closer to the (key,value) pair than its own key. A cache lookup will happen if at least one of these nodes is of the appropriate color. However, since $\alpha = 1$, we look in a correct colored cache with the following probability: $P_{\text{lookup}} = (1 - (1 - \frac{1}{C})^k)$.

The find value process ends as soon as a lookup returns a cached result. Hence, we can conclude that the find value operation will stop at any given step with the following probability $P_{\text{FindValueHit}} = P_{\text{lookup}} \cdot P_{\text{hit}}$.

Back to the previous example where $P_d = 0.01$ and $C = 17$, the chances of a cache hit when using a local cache is only 1% on each step of the find value process. In contrast, for colored caches this increases to a 5.4% hit rate per step.

When $\alpha > 1$, there are $\alpha$ concurrent requests. Whenever one of the messages returns, or times out, another request is issued. Yet, to simplify the analysis, we assume that the replies to all $\alpha$ requests return exactly at the same time with k new nodes for all the requests together. Thus, on each step we send $\alpha$ concurrent requests. In this case, as long as there are $\alpha$ or more correctly colored nodes in the k-candidate list we will make $\alpha$ lookups, and if there are fewer than $\alpha$ correctly colored nodes we will make fewer than $\alpha$ lookups.

To properly handle this situation, we define the following: $P_{j-\text{lookup}}$ is the probability that there are exactly $j$ correctly colored nodes in the candidates list. $P_{j>\text{lookup}}$ is the probability that there are more than $j$ correctly colored nodes among the candidates. Using these definitions and our assumptions we conclude that:

$P_{1-\text{lookup}} = \binom{k}{1}((1 - \frac{1}{C})^{k-1}(\frac{1}{C}))$

$P_{2-\text{lookup}} = \binom{k}{2}((1 - \frac{1}{C})^{k-2}(\frac{1}{C})^2)$

$P_{j-\text{lookup}} = \binom{k}{j}((1 - \frac{1}{C})^{k-j}(\frac{1}{C})^j)$

$P_{j>\text{lookup}} = \sum_{i=j+1}^{k} P_{i-\text{lookup}}$

Using this terminology we can conclude that in case $\alpha > 1$, the following holds under our assumptions:

$P_{\text{FindValueHit}} = \sum_{i=1}^{\alpha} P_{i-\text{lookup}} \cdot (1 - (1 - p_{\text{hit}})^i) + P_{\alpha>\text{lookup}} \cdot (1 - (1 - P_{\text{hit}})^\alpha)$.

Continuing our example, if we use a concurrency level of $\alpha = 3$, we will now have a 7% chance to obtain a cache hit, while a local cache alone gives only about 3% of a cache hit.
6.6.1.6 Expected message cost of iterative find value:

In our scheme, after the forwarding phase ends, the node starts an iterative find value process. This process only starts if the accepting node has no cached copy initially and it ends either when encountering a cached copy or when enough steps were taken to reach the value in Kademlia. At each lookup step $\alpha$ concurrent messages are sent out and we count each one twice since it requires both the remote node to handle the message and the local node to receive a reply.

Denote $D$ the distance in the Kademlia XOR metric between the original node $q$ to a key $d$ and denote $\delta(D)$ the average number of steps a cache-less Kademlia will take to find $d$. We will now estimate the expected message cost of the iterative find value phase as a function of $\delta(D)$, $p_d$ and the length of the forwarding path $L_f$.

$$\text{IterativeFindValueCost}(\delta(D), p_d, L_f) = 2\alpha \cdot P_{\text{miss}}(\sum_{j=1}^{\delta(D)} - L_f \cdot (1 - P_{\text{FindValueHit}}(p_d)))^{j-1}.$$  

$$P_{\text{FindValueHit}}(p_d) + (\delta(D) - L_f)(1 - P_{\text{FindValueHit}}(p_d))^{\delta(D)}$$

6.6.1.7 Expected message cost of find value:

Summing up what we already know, we can now estimate the total cost of a find value operation in Kaleidoscope:

$$\sum_{i=1}^{\delta(D)} (P(L_f = i) \cdot \text{ForwardingCost}(i) + \text{IterativeFindValueCost}(\delta(D), p_d, L_f))$$

6.6.2 Load Distribution Between the Colors

Ideally, all nodes should receive an equal load regardless of their color. However, since the request distribution is not necessarily uniform, the load experienced by nodes of different color is not equal. Below, we estimate the likelihood for deviation from the average expected load experienced by the nodes of a given color.

We associate each key $d$ with a random variable whose popularity is taken from the original distribution. Hence, $E(d) = P_d$. Using these random variables, we define new random variables $\{C_i\}$ to be the summation of all random variables of the color $i$. An immediate observation is that for any two colors $i, j$, $E(C_i) = E(C_j)$. Since for every splitting that favors one of the colors over the other, we can change the color name and receive the opposite one. Let $T$ to be the total number of find value requests in the system so far. $T$ Hence, $E(C_i) = E(C_j) = \frac{T}{C}$ for every $i, j \in [C]$. By applying Markov’s inequality, we conclude that: $P(C_i > \beta E(C_i)) < \frac{1}{\beta}$ and therefore: $P(C_i > \beta \frac{T}{C}) < \frac{1}{\beta}$. This result limits the probability that a certain color is significantly more popular than other colors. In particular, the probability that a specific color is $\beta$ times more popular than the average is only $\frac{1}{\beta}$. To guarantee that this property holds for any key popularity distribution only random hash functions should be employed.
6.7 Performance Measurements

In this section, we evaluate the performance costs and benefits of Kaleidoscope. We also compare the performance of Kaleidoscope to the one obtained with a local cache as proposed in [15] (a.k.a. Local) as well as the caching scheme suggested by the original Kademlia paper [30] (a.k.a. KadCache). Notice that our work is the first to report performance measurements for KadCache. We examine both the overall system performance under a variety of workloads including uniform and skewed value popularity distributions, and the load placed on the busiest nodes in the system.

For the evaluation, we have used a Java implementation of Kaleidoscope, KadCache, and Local. We have experimented with several different sizes of networks by running multiple Java VMs (one VM per 80 nodes) on a single computer and emulating the users find value requests (we measure message count and not latency). These requests are picked at random from a given distribution. For this, we used synthetic distributions, distributions directly taken from a real YouTube data set [12], and a real Wikipedia data set [36]. In the synthetic distributions, each node in the system periodically picks an item out of 100,000 possible keys according to the specific distribution and issues a find value request for that key. In the YouTube distributions, we used a data set that contains statistics of over 161k newly created videos. These videos were monitored weekly during 21 weeks starting from 16th April, 2008. We used the number of views per week in order to directly generate a distribution that reflects the popularity of each video during that week.

The Wikipedia dataset, it is very extensive contains a trace of 10% of all user requests issued to Wikipedia (in all languages) during the period between September 19th 2007 and October 31th 2007. Unfortunately, the trace is recorded from the server side and does not contain client information. We therefore simply picked a continuous slice of 1.75 million requests, cut it into equal chunks and distributed to all nodes. Each request is then assigned to a key and is searched for.

In all experiments, caches are given a warm-up period in which each node in the system issues 200 find value requests. After the warm-up period, each node in the system issues 500 additional find value requests. Statistics of message send/receive incoming/outgoing bandwidth are monitored locally by each node and are collected by HTTP at the end of the experiment. Our experiments where performed with the following parameters: bucket size $k = 7$; network sizes: 100, 500, 1, 500 and 2, 500 nodes; request distributions: Zipf 0.9, Zipf 0.7, and uniform. Zipf distributions with similar values were found, e.g., in Web caching and file sharing applications [7, 19]. Notice that in the case of 2, 500 nodes, the experiment includes a warm-up period of 500, 000 requests followed by additional 1, 250, 000 requests.
6.7.1 Mathematical Analysis Evaluation

In order to evaluate our theoretical analysis, we have conducted two experiments with uniform key request distribution and 1500 peers and compared the message cost obtained from the actual runs of Kaleidoscope with the ones predicted in the analysis. Yet, there is one parameter of the formal analysis, $\delta(D)$ (the expected number of steps Kademlia would have invoked to find a given value), for which we have no analytical expression. Hence, we have computed $\delta(D)$ empirically by running plain Kademlia multiple times in the same setting.

In the experiments, all the nodes stop every 50 requests (total of 75,000 find value operations) and gather the total number of messages handled in the system, excluding ping messages that are not part of a find value process. The total number we calculated is then divided to the actual number of find value operations to yield an average find value cost. The results of these experiments are reported in Figure 6.2. As can be seen, the results predicted by the formal analysis are fairly close to the ones obtained in practice. Let us emphasize that the “handled messages” results as presented in Figure 6.2 and elsewhere in this chapter count all messages sent in the system – there are no additional hidden overheads!

6.7.2 Performance Overview

Figure 6.3 presents the load distribution on the nodes in the system. It compares plain Kademlia (without caching), Kaleidoscope without caching and Kaleidoscope with a size 100 cache. As can be seen, the load distribution of Kademlia is highly skewed in these experiments. This is caused both from an uneven load distribution and from an uneven representation in routing tables. We can see here how Kaleidoscope helps alleviate this. When Kaleidoscope runs without a cache, we see a significant reduction of the load placed upon the hot nodes. This is due to our hybrid forward/iterative approach and the fact that in Kaleidoscope overloaded nodes can send NACKs in order to reduce their representation in routing tables. A further significant reduction is obtained through the colored cache.

Interestingly, even a small 100 items LRU cache has a dramatic impact on the performance of Kaleidoscope. This is achieved by our augmented routing scheme, which favors contacting nodes that are likely to hold a cached copy.

Another important issue of any caching scheme is its warm up period. Since session lengths in P2P systems are relatively short, large caches that require long warm up periods to operate are undesirable. Hence, we focus most of our measurements on small, easily warmed up 100 items caches. The expected performance of our caches during the warm up phase is somewhere between the no cache behavior and the warmed cache behavior. As can be observed, both offer significant improvement to the load distribution.
6.7.3 Effect of Overload Protection

Figure 6.4 exhibits the effect of our overload protection mechanism on the message cost of Kaleidoscope for both 7 and 17 colors. Recall that overload protection enables busy nodes to reduce their incoming bandwidth by reducing their level of representation in routing tables. As can be seen in Figure 6.4, indeed the overload protection scheme significantly reduces the load on the busiest nodes. The overall system load slightly increases as well (in our experiments by around 1.5%), but the load on the “hottest” nodes was reduced by as much as 25% in the case of 17 colors. Also, this optimization is more effective with 17 colors than with 7 since in the former, the chances of hitting a cache of the correct color are lower making the forwarding phase longer.

Figure 6.2: Comparing the actual find value cost to the mathematical estimation.

6.7.4 Different Color Configurations

Increasing the number of colors increases the forwarding path length and the cache hit rate, yet it takes longer to reach a cache. In workloads with in efficient caching, more colors yields better performance. However, in workloads with efficient caching, a small number of colors can be better since we reach a cache sooner. In addition, under a skewed distribution the load on colors is not well balanced. In particular, increasing the number of colors too much creates a new kind of hot nodes, nodes with a hot color.

In Figure 6.5, we can see that indeed 17 colors seems better than 7 colors. However, further increasing the number of colors to 34 yields little improvement in the average lookup cost but no improvement on the load distribution. Since short forwarding phases have significant benefits (see Section IV-D), we continue our evaluation focusing on the 17 colors configuration.
6.7.5 Kaleidoscope vs. Local and KadCache

Figure 6.6 compares Kaleidoscope (with 17 colors) vs. Local and KadCache when using 100 items caches under various request distributions. As can be seen, Kaleidoscope outperforms Local and KadCache both in terms of the overall load and in terms of the load on the busiest nodes. When comparing Local and KadCache, Local is better than KadCache in terms of hot spot handling while KadCache is better than Local in terms of the overall load.

A summary of the average message cost of the caching schemes, Kaleidoscope, Local, and KadCache, running with a bounded cache of 100 items, normalized to the cost of the original (cache-less) Kademlia appears in Figure 6.7. It shows that all caching strategies improve with the network size. Yet, Kaleidoscope is better than the others, and in particular is quite effective even in very small networks.

In Figure 6.8, we compare the behavior of the three caching schemes with varying cache sizes. As can be seen, when increasing the cache size from 100 to 800 objects, the message cost is somewhat reduced due to an improved cache hit rate. Still, Kaleidoscope does better with a 100 objects cache than any of the other schemes with an 800 objects cache.
6.8 Conclusions

In this chapter, we have presented Kaleidoscope, an augmented caching and routing scheme that greatly reduces the hot spots problem in Kademlia while also lowering its total communication overhead. We have presented a formal analysis of the expected behavior of Kaleidoscope alongside an experimental performance evaluation of our implementations of the original Kademlia (including previously proposed caching optimizations) and Kaleidoscope. The findings of this study exhibit that the formal analysis is indeed indicative of the performance of the system and that Kaleidoscope satisfies its goals. The code of Kaleidoscope is available in open source, and is integrated as part of our OpenKad project [1]. The test package and traces used in this chapter is available to the public as well.

Kademlia has become so prevalent in popular P2P applications due to its scalability and robustness to churn. One of the main reasons for this is Kademlia’s dynamic multiple selection routing process, unlike many other KBRs (DHTs) like Chord [34], which are rigid. Yet, this is also the reason why effective caching in Kademlia is more difficult to achieve. As we have shown, our scheme provides an effective caching solution to Kademlia, without imposing rigidity on its routing process. It would be interesting to apply our ideas to additional KBRs that offer at least some routing flexibility such as Pastry [33].

Looking ahead, we would like to optimize Kaleidoscope for low latency, by employing proximity awareness routing techniques. We would also like Kaleidoscope to be able to dynamically pick a routing strategy (recursive/iterative) by monitoring the

Figure 6.4: Effects of overload protection algorithm on the load placed on 500 busiest nodes (2,500 nodes, YouTube).
Figure 6.5: Distribution of handled messages (sorted histogram), 7 colors, 17 colors and 34 colors (2,500 nodes).

current network churn rate and the freshness/proximity of nodes in its routing tables.
6.8 Conclusions

![Figure 6.6: Distribution of handled messages (sorted histogram), Kaleidoscope vs. KadCache vs. Local Cache with a 100 items cache and 2,500 nodes.](image)

<table>
<thead>
<tr>
<th>Dist/Size</th>
<th>Local 100</th>
<th>Local 500</th>
<th>Local 1500</th>
<th>KadCache 100</th>
<th>KadCache 500</th>
<th>KadCache 1500</th>
<th>Kaleidoscope 100</th>
<th>Kaleidoscope 500</th>
<th>Kaleidoscope 1500</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform</td>
<td>0.99</td>
<td>0.98</td>
<td>0.99</td>
<td>1.04</td>
<td>0.98</td>
<td>0.94</td>
<td>0.96</td>
<td>0.85</td>
<td>0.84</td>
</tr>
<tr>
<td>Zipf 0.7</td>
<td>0.96</td>
<td>0.91</td>
<td>0.38</td>
<td>0.97</td>
<td>0.85</td>
<td>0.32</td>
<td>0.64</td>
<td>0.69</td>
<td>0.27</td>
</tr>
<tr>
<td>Zipf 0.9</td>
<td>0.76</td>
<td>0.76</td>
<td>0.73</td>
<td>0.79</td>
<td>0.69</td>
<td>0.63</td>
<td>0.62</td>
<td>0.53</td>
<td>0.50</td>
</tr>
</tbody>
</table>

![Figure 6.7: Normalized message cost of the three caching schemes W.R.T the original (cache-less) Kademlia - lower is better](image)
Figure 6.8: Kaleidoscope vs. KadCache vs. Local Cache when changing the cache size
7 Discussion

Throughout the short history of online social networks, we have seen countless incidents of data theft and abuse. Some were done maliciously by hackers infiltrating the cloud. Others were done by a simple human error or invalid configuration. Thus, we argue that one possible way of keeping our data safe is by decentralizing and encrypting it. Decentralization avoids having one single quality target while encryption prevents unauthorized users from reading it.

In E-Wolf, we can see that the entire system was designed to be decentralized. The communication layer is built upon a well known and widely used Peer to Peer routing scheme. It provides basic anonymity along side a reasonable performance overhead. The storage system is self organizing and requires no complicated central communication. Using several replicas of our data we achieve a high bandwidth communication by parallelizing our connections and downloading different parts of it from several hosts simultaneously. This ability greatly improves our performance comparing to downloading from a single source that may be located far away.

7.1 Future work

Although all E-Wolf components are implemented, there are numerous features missing and countless ways to improve their performance. In this section we suggest a few critical features to focus on.

7.1.1 Communication Layer

In recent years the popularity of hand held devices has skyrocketed. These devices present new challenges for Peer to Peer communication whereas their availability changes rapidly. Whether the user enters an elevator or simply drives through a tunnel, the device is going to disconnect from the network and reconnect shortly after. This may cause a very high churn rate for the DHT that might not be able to handle it. Moreover, due to the sheer number of these devices, cellphone network providers tend to use NAT in IP address allocation. The use of NAT renders the devices behind it unreachable from the outside. All connection outside the NAT domain are initiated from the device and not to it.

We have seen an attempt of solving this problem through the use of a cloud [35]. However, a cloud has many disadvantages of its own, such as being a quality target
and a single point of failure. We suggest the development of a NAT oriented Peer to Peer network. That is, a self organizing network that will support large amount of its nodes being behind NAT. One way to do so is to use a hierarchy of nodes such that some function as proxies to several others behind NAT.

Another interesting improvement to the Peer to Peer network may be the application of Geo-aware distance metric. Instead of using arbitrary IDs for nodes and defining the distance between them as the result of XOR calculation on their IDs, one might design a more sophisticated metric function. The new function should match the latency between nodes instead. Thus, improving performance of peer discovery by querying low latency nodes first.

7.1.2 Storage Layer

As we have seen, ChunKeeper uses a Tit-For-Tat algorithm for determining the ratings of other nodes. The rating function, by which we make our decisions, is still very rough. Fine tuning it using a real-life data set of nodes storage patterns may very well hold the difference between a successful storage system and a completely useless one. Should the rating function be too generous, the amount of freeloaders may overrun the system. However, if it is too strict, new relationship spawning will be hampered, rendering the replication level very low or even local only.

7.1.3 Social Network Layer

In this layer we may have the greatest number of small improvements that are not critical. We chose to focus on the one feature E-Wolf lacks: instant messaging. The possibility to engage in a real-time conversation with one or several other users has become a crucial requirement for the success of an online social network. As we can see, some social networks provide only this feature and are considered very successful. In implementing this feature security precautions must be considered. All messages must be encrypted and secret key exchange must be provided.
Bibliography


אב-א: רשת 버טית מבוורת

ב머שת

איל קובר
אבר–א: רשת הבריתים מבויתת
בmarshahת

תיוור על פרוייקט

לשים מילוי חלקי של הדרישות לקבלת תואר
منهجי сразועים במדעי המחשב

יאל קיבר

הוגש למט של הטכניון – מכון טכנולוגי לישראל
אדר א' התשע”ד
חיפה
פברואר 2014
הפרוייקט נעשה בהנחיית פרופסור רועי פרידמן לפקולטה למדעי המחשב.

אני מודה לפניך על ההמענה והифика המתנדבים בהשכלתי.
E-Wolf היא מערכת בעלתデザイン מודולרי שבבה הפונקציונליות המורכבות של כל רכיב הוערכו מחולים למODULEים שבכל אחד מאומן של מODULEים משותפים בפונקציונליות של אלו שלשתן. עניב זה מאופה התכונות הדומות של המODULEים ומאופי ניール התכונות וה مشيراות.

באקוף ספציפי יותר, ב E-Wolf כל זן מתמט מתמודד עם עלה לנמה. המODULEים הכלליים בおります מאופי

יוניל גלייטרים וטורבר תועש. תופעת עלי, תופעת וטורבר תועש מתאימה לטרים של מODULEים של בעהל.

עם נמיש על התאום בין מתאומי חלון וחלון פמוקים ומODULEים של בעהל של עלי הירושמות. עליה הרוחב עליי התאומים העילויים בוחר כVentaוס עליי החש 제ורחה המODULEים של בעהל וтверждаו עליי תמים. כמ', כיווה

ניאוש טסינ שמש על התאומים של הנוסח בידות נטוש המODULEים בידות האופי

E-Wolf הוא מים ב Java, שזאת זן צפיות קוד פותח. למישהו, כל הידמתמודול שליה אוף צפיות

נפרד. פוריקט טסינ אוף מאפיournim שמש בחלקים של שמי של E-Wolf בפוריקט תחרים, ומעודד את

פוטות מODULEים רפים עליי פונקציונליות שלעה לטרמכת.
תקציר

בשנים האחרונות, רשתות חברתיות באינטרנט הפכו לחלק בלתי נפרד מהחיים. רב מの中ים משתמשים במשתמש החברתיות הממאמזות והבנקודות המרכזיות לאפשר נגישות ושיתוף קבוצתי של榕שים פורסמים. פיסוקי לינקדיני, ומפקחים על מספרםםויה של יוזמות ופרסומים של משתמשים נוספים. mutual denials של השבטים האסטרטגיים של היכולים אביטל בויתוקןган נותרו עם פיתויים של יוזמות ופרסומים של משתמשים נוספים. mutual denials של השבטים האסטרטגיים של היכולים אביטל בויתוקןган נותרו עם פיתויים של יוזמות ופרסומים של משתמשים נוספים. mutual denials של השבטים האסטרטגיים של היכולים אביטל בויתוקןган נותרו עם פיתויים של יוזמות ופרסומים של משתמשים נוספים. mutual denials של השבטים האסטרטגיים של היכולים אביטל בויתוקןган נותרו עם פיתויים של יוזמות ופרסומים של משתמשים נוספים. mutual denials של השבטים האסטרטגיים של היכולים אביטל בויתוקןган נותרו עם פיתויים של יוזמות ופרסומים של משתמשים נוספים. mutual denials של השבטים האסטרטגיים של היכולים אביטל בויתוקןган נותרו עם פיתויים של יוזמות ופרסומים של משתמשים נוספים. mutual denials של השבטים האסטרטגיים של היכולים אביטל בויתוקןган נותרו עם פיתויים של יוזמות ופרסומים של משתמשים נוספים. mutual denials של השבטים האסטרטגיים של היכולים אביטל בויתוקןган נותרו עם פיתויים של יוזמות ופרסומים של משתמשים נוספים. mutual denials של השבטים האסטרטגיים של היכולים אביטל בויתוקןган נותרו עם פיתויים של יוזמות ופרסומים של משתמשים נוספים. mutual denials של השבטים האסטרטגיים של היכולים אביטל בויתוקןган נותרו עם פיתויים של יוזמות ופרסומים של משתמשים נוספים. mutual denials של השבטים האסטרטגיים של היכולים אביטל בויתוקןган נותרו עם פיתויים של יוזמות ופרסומים של משתמשים נוספים. mutual denials של השבטים האסטרטגיים של היכולים אביטל בויתוקןган נותרו עם פיתויים של יוזמות ופרסומים של משתמשים נוספים. mutual denials של השבטים האסטרטגיים של היכולים אביטל בויתוקןган נותרו עם פיתויים של יוזמות ופרסומים של משתמשים נוספים. mutual denials של השבטים האסטרטגיים של היכולים אביטל בויתוקןган נותרו עם פיתויים של יוזמות ופרסומים של משתמשים נוספים. mutual denials של השבטים האסטרטגיים של היכולים אביטל בויתוקןган נותרו עם פיתויים של יוזמות ופרסומים של משתמשים נוספים. mutual denials של השבטים האסטרטגיים של היכולים אביטל בויתוקןган נותרו עם פיתויים של יוזמות ופרסומים של משתמשים נוספים. mutual denials של השבטים האסטרטגיים של היכולים אביטל בויתוקןган נותרו עם פיתויים של יוזמות ופרסומים של משתמשים נוספים. mutual denials של השבטים האסטרטגיים של היכולים אביטל בויתוקןган נותרו עם פיתויים של יוזמות ופרסומים של משתמשים נוספים. mutual denials של השבטים האסטרטגיים של היכולים אביטל בויתוקןган נותרו עם פיתויים של יוזמות ופרסומים של משתמשים נוספים. mutual denials של השבטים האסטרטגיים של היכולים אביטל בויתוקןган נותרו עם פיתויים של יוזמות ופרסומים של משתמשים נוספים. mutual denials של השבטים האסטרטגיים של היכולים אביטל בויתוקןган נותרו עם פיתויים של יוזמות ופרסומים של משתמשים נוספים. mutual denials של השבטים האסטרטגיים של היכולים אביטל בויתוקןган נותרו עם פיתויים של יוזמות ופרסומים של משתמשים נוספים. mutual denials של השבטים האסטרטגיים של היכולים אביטל בויתוקןган נותרו עם פיתויים של יוזמות ופרסומים של משתמשים נוספים. mutual denials של השבטים האסטרטגיים של היכולים אביטל בויתוקןג
The document is in Hebrew and discusses the concept of Diaspora and the creation of social networks that aim to separate the technological infrastructure that powers them from the social networks they serve. It mentions technologies such as Safebook for social networking, which allows users to manage their data securely. The document also talks about the need for standard protocols to ensure compatibility and safety, and the importance of providing users with access to their information. It notes that while this may be challenging, it is essential for the future of social networking. The document concludes by discussing the architecture of E-Wolf, which is used as an example to illustrate the principles discussed.
E-Wolf

משתמש ב. בפרק 5 אנו מתארים את האינטראקציות החברתיות של E-Wolf. 

אנו מגדירים בפרוק תיאור בצורה מפורטת של_EMITת שבל E-Wolf. בפרק 7 אנו מציגים עבודה לעודד עמידת

E-Wolf לפגוש עידן

לפיו.