ROYAL INSTITUTE
OF TECHNOLOGY

Kademlia on the Open Internet

How to Achieve Sub-Second Lookups in a Multimillion-Node DHT Overlay

RAUL JIMENEZ

Licentiate Thesis
Stockholm, Sweden 2011

TRITA-ICT/ECS AVH 11:10
ISSN 1653-6363
ISRN KTH/ICT/ECS/AVH-11/10-SE
ISBN 978-91-7501-153-0

KTH School of Information and
Communication Technology
SE-164 40 Stockholm
SWEDEN

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av Communication Systems fredag den 9 december 2011 klockan 10.00 i C2, Electrum, Kungl Tekniska högskolan, Forum, Isafjordsgatan 39, Kista.

© Raul Jimenez, December 2011
This work is licensed under a Creative Commons Attribution 2.5 Sweden License.
http://creativecommons.org/licenses/by/2.5/se/deed.en

Tryck: Universitetsservice US AB

Abstract

Distributed hash tables (DHTs) have gained much attention from the research community in the last years. Formal analysis and evaluations on simulators and small-scale deployments have shown good scalability and performance.

In stark contrast, performance measurements in large-scale DHT overlays on the Internet have yielded disappointing results, with lookup latencies measured in seconds. Others have attempted to improve lookup performance with very limited success, their lowest median lookup latency at over one second and a long tail of high-latency lookups.

In this thesis, the goal is to to enable large-scale DHT-based latency-sensitive applications on the Internet. In particular, we improve lookup latency in Mainline DHT, the largest DHT overlay on the open Internet, to identify and address practical issues on an existing system. Our approach is implementing and measuring backward-compatible modifications to facilitate their incremental adoption into Mainline DHT (and possibly other Kademlia-based overlays). Thus, enabling our research to have impact on a real-world system.

Our results close the performance gap between small- and large-scale DHT overlays. With a median lookup latency below 200 ms and a 99th percentile of just above 500 ms, our median lookup latency is one order of magnitude lower than the best performing measurement reported in the literature. Moreover, our results do not show a long tail of high-latency lookups, unlike previous reports.

We have achieved these results by studying how connectivity artifacts on the underlying network —probably caused by firewalls and NAT devices on the Internet— affect the DHT overlay. Our measurements of the connectivity of more than 3 million nodes reveal that connectivity artifacts are widespread and can severely degrade lookup performance.

Scalability and locality-awareness have also been explored in this thesis, where different mechanisms have been proposed. Some of the mechanisms are planned to be integrated into Mainline DHT in future work.

A mis padres, a quienes quiero y admiro

Acknowledgements

This thesis is the result of my work in collaboration with Björn Knutsson and Flutra Osmani. While I claim most of the contribution as my own, this could not have been possible without their hard work and support. I cannot write about *my work* in good conscience. That is why I will use *we* instead of *I* when describing *our work* throughout the thesis.

I thank Björn Knutsson, who has gone well beyond his obligations as advisor to support my research and establish a working environment based on respect and passion for high spirited discussions. Flutra Osmani also deserves a good deal of gratitude for her collaboration in most of the research in this thesis.

Many thanks to all my colleagues at TSLab, with whom I have had interesting discussions about my research and a great deal of fun.

I am grateful to Björn Pehrson, a good advisor who is so passionate about his work that is hard to notice that he has retired.

I thank Seif Haridi, who recently took his role as advisor when Björn Pehrson retired. He has been very supportive and I look forward to working with him and his colleagues at SICS.

Thanks to all my maser thesis students: Jorge Sainz Raso, Flutra Osmani, Sara Dar, Ismael Saad Garcia, Shariq Mobeen, S.M. Sarwarul Islam Rizvi, Zinat Sultana, and Md. Mainul Hossain. I hope you have learned something from me, I certainly have learned a lot from you.

Urko Serrano and Pehr Söderman took the time and effort to read parts of the draft. Thanks for your comments.

I sincerely appreciate the effort of Arnaud Legout of INRIA and Jim Dowling of SICS to make room in their busy agendas to be opponent and internal reviewer, respectively.

Thanks to Amir Payberah and Fatemeh Rahimian who helped me with the draft and paperwork. And talking about paperwork, thanks to Marianne Hellmin, whose diligence made all the administrative procedures much easier for me.

Thanks to my colleagues in the P2P-Next project. It is great to collaborate with such smart and friendly people. Special thanks to Lars-Erik of Dacc; and Victor Grishchenko, Johan Pouwelse, and Henk Sips of TUDelft with whom I collaborated to produce two papers, which are not included in this work.

Finally, I thank my family and friends for their unconditional support. Especially Svetlana Panevina, my wife, my happiness.

Errata

In papers [1], [2] and [3] (acknowledge section), the project number should be 212617 instead of 21617. Papers [1] and [2] are reproduced in their original form in Chapters 7 and 8, respectively.

Contents

Contents ix

I Thesis Overview 1

1 Introduction 3

2 Background 11
- 2.1 P2P-Next's Fully-Distributed Peer Discovery System . . . 11
- 2.2 Distributed Hash Tables . . . 12
- 2.3 Mainline DHT . . . 15
- 2.4 Related Work on Improving DHT Performance . . . 16

3 Problem Definition 17

4 Thesis Contribution 19
- 4.1 List of Publications . . . 19
- 4.2 Scalability and Locality-Awareness . . . 20
- 4.3 Connectivity Properties . . . 20
- 4.4 Sub-Second Lookups on a Multimillion-Node DHT Overlay . . . 20
- 4.5 Source Code . . . 21
- 4.6 Individual Contribution . . . 21

5 Discussion 23

6 Conclusion 25
- 6.1 Future Work . . . 26

Bibliography 27

II Research Papers **31**

7 CTracker: a Distributed BitTorrent Tracker Based on Chimera **33**

8 Connectivity Properties of Mainline BitTorrent DHT Nodes **43**

9 Sub-Second Lookups on a Large-Scale Kademlia-Based Overlay **55**

Part I

Thesis Overview

Chapter 1

Introduction

Scaling services used to be as simple as replacing one server with a more powerful one. At first, improvements in processing speed and caches were enough. Then, servers started to incorporate multiple processors, and even multiple multi-core chips.

At some point, a single powerful server was not enough to meet demand for popular services and *clusters* of tightly coupled servers were created. Then, *GRID technology* spread, allowing more loosely coupled, heterogeneous, and geographically dispersed systems. Finally, we currently witness the dramatic raise of *cloud computing*.

Cloud computing providers deploy inter-connected massive data centers, each one consisting of hundreds or thousands of inexpensive general-purpose machines. These large-scale distributed systems are possible thanks to automated mechanisms that coordinate all these machines. Examples of such mechanisms are: Amazon's Dynamo [4], Google's MapReduce [5], and Microsoft's Dryad [6].

An alternative to —or an option to reduce load on— data centers is to outsource some of the computation work to machines owned and operated by the users of the service. A remarkable example of this model is the SETI@home project [7], where millions of users contribute to the search for extraterrestrial intelligence.

Commercial companies have also leveraged user's resources to scale their services and reduce costs. Well-known examples are: Skype[1](video-conference service) and Spotify [8] (music streaming service). In these systems, users' machines collaborate directly with each other using *peer-to-peer* protocols.

Peer-to-Peer Systems

Peer-to-peer (P2P) systems, the subject of this thesis, are composed by machines which both contribute and consume resources —i.e., they simultaneously act as servers and clients for the same service. These machines are called *peers* because

[1]`http://www.skype.com/` (Oct. 2011)

they have equal responsibility in the system. Ideally, each peer adds resources (increasing scalability) and individual peer failures do not cause a system failure (the P2P system is robust to churn).

One of the main challenges in P2P systems is to coordinate peers in a way that peers demanding resources are able to find peers offering these resources. The simplest approach is to coordinate all these peers from a centralized service. In this case, the system behaves like an orchestra where the *coordination service* fills the role of the orchestra's conductor and the rest gracefully follow the conductor's instructions.

A video streaming service, for instance, could be implemented in this fashion. Each peer interested in a given video stream contacts the *coordination service*, which provides the network address of other peers from where the video stream can be requested. Peers consuming the video stream also contribute resources to the system by forwarding the video stream to other peers, thus scaling the service and reducing the content provider's cost. The *coordinator machine* keeps a global view of the system, matching available resources with demand.

This centralized coordination mechanism creates a single critical point of control. While there are maintenance costs and technical issues (namely, scalability and single point of failure) associated with centralized services; there are other reasons for service providers to prefer a centralized mechanism. For some commercial companies, for instance, features such as tighter control and easier monitorization may outweight the potential benefits of a decentralized mechanism.

Peer-to-Peer Communities

The rise of P2P technologies not only enabled companies to scale their services while reducing their cost, but also provided individuals with the means to build *distributed communities* with no central point of control. The most visible of those are the communities surrounding some popular *file sharing* systems.

Early P2P file-sharing systems were divided in two types: centralized coordination and fully-distributed. The first type imposes a centralized element —Napster's central index [9], BitTorrent tracker [10], eDonkey/eMule's *servers* [11]— in an otherwise distributed system.

There were also early attempts to build large-scale fully-distributed systems based on unstructured overlays (Freenet [12], Gnutella [9, 13]) but performance [14] and scalability [15] issues made them uncompetitive compared to BitTorrent and eMule.

It is only now that distributed hash tables (DHTs), introduced later in this chapter, offer a third alternative in which those critical centralized services can be distributed among peers, while keeping the rest of the system unmodified. In particular, both eMule and BitTorrent have evolved to combine their high-performance data transport protocols with fully-distributed *coordination services* based on large-scale DHT overlays.

There are, however, unresolved issues with large-scale DHT overlays. This thesis will mainly focus on the performance issue and also study scalability and locality-awareness.

Low-Latency DHT on the Open Internet

The main goal of this thesis is to understand and, if possible, close the substantial performance gap between DHT overlays in the lab and large-scale DHT overlays on the open Internet.

Our approach is adapting one of the existing large-scale DHT overlays on the Internet. These adaptations are the fruit the analysis of our extensive experiments performed on the actual multi-million-node DHT overlay.

Our results close the performance gap, enabling latency-sensitive services on large-scale DHT-based systems on the Internet. In particular, our code is used by a fully-distributed content streaming platform whose responsiveness should be able to compete with cable and satellite TV latencies. This platform is currently developed by the P2P-Next project and will be further discussed in Section 2.1.

Distributed Hash Tables

Distributed hash tables (DHTs), as the name suggests, provide the functionality of hash tables —i.e. *store* and *retrieve* operations. The critical difference being that a DHT maps *keys* to nodes' addresses instead of memory locations. That is, the DHT provides a mechanism to find the node responsible for handling *store* and *retrieve* operations for a given key.

A *DHT overlay* consists of nodes connected to each other in a particular structure to be able to efficiently perform store/retrieve operations. It is called overlay because a DHT overlay can route messages from one node to another independently of the underlying network, which provides the basic connectivity between nodes. The Internet is an example of an underlying network used by DHT overlays.

In a DHT overlay, each node keeps a routing table containing pointers to other nodes, which are called neighbors. Whenever a node needs to perform a DHT operation —for instance, store a value for a given key— that node needs to locate the node responsible for the storage of that key. This process is called *lookup* and consists of a number of messages being routed with the help of the nodes' routing tables, getting closer to the responsible node at each step.

Different DHT designs propose different approaches to build DHT overlays. Two important properties are overlay geometry and routing [16]. Geometry (e.g. ring, tree) is determined by how neighbors are selected. Routing can be recursive or iterative. These properties will be further discussed in Chapter 2.

Regardless of their geometry and routing, DHTs have been designed to be self-organized, scalable, and robust to churn. Their **self-organizing** nature removes the necessity of a central point of control (and failure). A **scalable** system is

able to grow its number of participants with a limited performance degradation. Finally, **robustness to churn** is the capacity to handle dynamic behavior, a critical property since nodes join and leave the DHT overlay independently of each other.

These properties have attracted much attention from researchers, who have proposed different DHT designs. Well-known DHT designs include: CAN [17], Chord [18], Pastry [19], Tapestry [20], and Kademlia [21]. All these DHT designs have been formally analyzed and tested in controlled environments such as simulators and small-scale deployments.

Deployment of DHT-based applications have received much less attention, though. Two remarkable exceptions are OpenDHT [22] and CoralCDN [23]. OpenDHT was[2] a DHT overlay that third-party applications could use to store and retrieve information. CoralCDN is a DHT-based CDN (content distribution network) where nodes act as proxies to a web server. CoralCDN is especially useful on the event of flash crowds when the web server is not able to cope with unexpected peak loads.

These two projects have run for several years (CoralCDN is still running), providing a great opportunity for researchers to discover and analyze issues related to real-world deployment. On the other hand, both of them where centrally managed (all the nodes were under the researchers' control) and the size of the DHT overlay was small (less than 500 nodes running on PlanetLab [24] in both cases [23, 25]).

Mainline DHT: the Largest DHT Overlay on the Internet

In this thesis, our main motivation stems from our interest in deploying large-scale DHT-based applications on the Internet. The Internet is a good candidate as *the* underlay network (layer providing connectivity between DHT nodes) because of the large number of devices it interconnects.

To our knowledge, only three deployed DHT overlays (all of them based on Kademlia [21]) consist of more than one million nodes: Mainline DHT (MDHT), Azureus DHT (ADHT), and KAD. The first two are independently used as trackers (peer discovery mechanisms) in BitTorrent [26], while KAD is used both for content search and peer discovery in eMule (a widely used file-sharing application).

For researchers, such deployments offer a unique opportunity to study large-scale distributed systems in a real-world environment. KAD is the overlay which has been studied most thoroughly [27–30]. ADHT [31, 32] and MDHT [31] have also been studied but not as much.

In this thesis, we focus on MDHT, the largest DHT overlay on the Internet. Jünnemann et al. [33] estimate its population is between five and ten million nodes[3].

By focusing on a single overlay, we aim to study in detail the behavior and issues of a real-world large-scale distributed system deployment. We argue that we need to understand the details that make deployment hard if we want DHTs to be

[2] Discontinued in 2009 (see `http://opendht.org`).

[3] A real-time estimation is available at `http://dsn.tm.uni-karlsruhe.de/english/2936.php` (October 2011).

a viable option to build large-scale distributed systems on the Internet. That is, we need to evaluate a deployed DHT overlay which, considering the abovementionated large performance gap, is clearly different from a simulated one.

Like in most of the studies on large-scale DHT overlays, improving lookup performance is our main quantitative goal. This is hardly surprising given the poor performance of such DHT overlays, far worse than measurements obtained on simulators [34, 35] and small-scale DHT overlays [25, 36], where lookup latency is measured in milliseconds.

The only measurement of lookup performance on MDHT we are aware of yielded a median lookup time of around one minute [31]. The best lookup performance ever reported on these large-scale DHT overlays is a median of 1.5 seconds, achieved by Steiner et al. [28] on KAD.

As discussed in Section 2.1, the kind of applications we target in this thesis cannot tolerate such high latencies due to their strict latency requirements. For example, one of these requirements could be: *"over 50% of the operations must perform within 500 ms and over 99% within one second"*.

Performance improvement is a central part of this thesis and will be addressed later in this chapter; but first, we need to understand the problem. For that, we analyze the potential causes of poor performance in Internet-deployed DHT overlays by defining and studying the underlying network's connectivity properties and their effects on the performance of the DHT overlay.

Connectivity Properties

Previous research indicated that connectivity artifacts on the underlying network may be the main cause of such poor performance in MDHT [31]. In their study, they found that a large fraction of the queries sent during a lookup are never responded, causing long delays. While others have proposed approaches to address this issue (e.g., reducing, or removing, timeout delays [32]), we see these failures (unresponded queries) as a mere symptom of a deeper problem.

The root of this problem, we argue in this thesis, is a mismatch between the underlying network's connectivity properties implicitly assumed by the DHT designers and the far-from-ideal connectivity properties actually present on the underlying network (the Internet in our case).

In Chapter 8, we characterize nodes' connectivity using three connectivity properties: **reciprocity**, **transitivity**, and **persistence**. Each of these properties is clearly defined and a mechanism to measure the connectivity properties of any node in the Mainline DHT overlay is presented.

In an ideal underlay, the connectivity between any two nodes would exhibit all three properties. Our measurement of over 3.6 million Mainline DHT nodes, however, revealed a large fraction of these nodes lacking one or more connectivity properties.

Given that DHT designs *implicitly assume* that the underlying network pro-

vides all three connectivity properties, we study the impact of the lack of each of these properties on the DHT overlay, concluding that these connectivity artifacts potentially degrade performance. In a previous work, Freedman et al. [37] studied how non-transitive connectivity in PlanetLab [24] degrades DHT performance.

Understanding the causes of MDHT's poor performance is just the first step towards improving lookup performance. The next step is to design mechanisms that address these issues and deploy them.

Chapter 8 describes the mechanisms we propose. A brief account of the deployment of some mechanisms is presented next.

Improving Performance on a Large-Scale DHT Overlay

Deploying modifications on a truly-distributed DHT overlay is possible but far from trivial. Unlike centrally-controlled DHT overlays such as OpenDHT and CoralCDN, Mainline DHT does not have an *authority* that can make global changes by pushing software updates to all nodes. Instead, the MDHT overlay consists of millions of computers running different BitTorrent clients developed by independent development teams; some implementations developed by the open-source community, others by commercial companies.

Therefore, it is possible to deploy modified nodes on the MDHT overlay because it is open to anyone, but it is very difficult to deploy global modifications which require the collaboration/synchronization of a number of independent development teams. One of the clear examples of the difficulty of deploying global backward-incompatible modifications is well illustrated by the IPv6 transition process which has proven to be a very hard task and still is far from complete.

While our long-term vision is to improve performance globally, our approach is to start with local backward-compatible modifications, in a bottom-up fashion. If we manage to significantly improve performance, while conserving other important properties, other MDHT developers may include our modifications into their software. For others to incrementally integrate these modifications, modifications must be simple and backward compatible.

In this thesis, we explore the approach of deploying nodes with modified routing table management and lookup algorithms; and evaluate the impact of these modifications on that node's performance. These local modifications are backward compatible. That is, they do not require any modification of the protocol nor the modification of existing nodes.

While we mainly focus on improving local performance, we also consider the impact of our modifications globally. Our modifications are designed not to degrade the performance of existing nodes. On the contrary, some of our modifications could potentially benefit existing nodes as well. Thus, as developers include our modifications in their clients, all nodes are potentially benefited. We elaborate on these local and global benefits in Chapter 9.

Scalability and Locality-Awareness

This thesis also considers adding locality-aware features to the DHT which would facilitate BitTorrent peers to find other peers within the same ISP, thus reducing BitTorrent traffic between ISPs. This traffic reduction in the inter-ISP links has the potential to reduce costs for ISPs. Varvello and Steiner [38] have recently studied this problem and proposed their own solution.

In addition, we proposed a simple mechanism which improves load balance in the DHT. This is our solution to the open scalability issue where nodes responsible for very popular keys have to bear much higher loads than average. Carra et al. [30] have recently proposed a different mechanism where ISPs inject locality-aware nodes in the Mainline DHT overlay.

Although we designed these mechanisms for Tapestry [20] (see Chapter 7), we consider integrating them into Mainline DHT as future work.

Thesis Contribution

The first mayor contribution is the performance improvement on an existing large-scale DHT overlay. The performance achieved is one order of magnitude better compared to previous attempts in the literature and closes the gap between performance observed in large-scale overlays versus simulations and small-scale overlays.

The second mayor contribution is the characterization of nodes' underlying connectivity properties, the analysis of the impact of each property on DHT performance, and the empirical connectivity characterization of over three million nodes in Mainline DHT.

Finally, we also propose a mechanism to both address scalability issues and add locality-aware features. We consider integrating these mechanism into Mainline DHT as future work.

We contribute all the software used in this thesis to the research community. This software includes all the tools and modules necessary to reproduce our results. Its open-source license allows others to adapt the tools to study DHT overlays.

Thesis Organization

The thesis is organized in two parts. The first part provides an overview and the second part is a compilation of peer reviewed papers.

The rest of the first part is organized as follows. The background is presented in Chapter 2. Chapter 3 defines the problem this thesis addresses. Chapter 4 summarizes the main contributions of this thesis. Chapter 5 discusses the direction and research focus of the work. Finally, Chapter 6 concludes and presents future work.

Chapter 2

Background

This chapter aims to introduce the necessary background to understand the rest of the thesis. The descriptions are by no means exhaustive but just provide a generic view of the concepts and are focused on the most relevant details related to this thesis work.

2.1 P2P-Next's Fully-Distributed Peer Discovery System

The work presented in this thesis is part of a large EU project called P2P-Next[1]. P2P-Next's main goal is to develop a fully-decentralized content distribution platform.

The P2P-Next platform (known as *NextShare*) has been built on top of BitTorrent. BitTorrent [26] is a widely-used peer-to-peer file-sharing platform where peers share content with each other.

Traditionally, BitTorrent has relied on centralized servers (called *trackers*) to coordinate peers. That is, trackers keep track of which peers participate in a given *swarm*.

Before going any further, let's define some terms used through out this thesis. The entities participating in the DHT overlay are called *nodes*, while those exchanging pieces of content using the BitTorrent protocol are *peers*. A *swarm* is a set of peers participating in the distribution of a given piece of content.

In an attempt to decentralize the tracker's tasks, a DHT-based mechanism was introduced in BitTorrent [39]. Currently, there are two independent DHT-based peer discovery mechanisms in BitTorrent: Mainline DHT and Azureus DHT. Both overlays are Kademlia-based. In this thesis, we focus on Mainline DHT, which has been briefly introduced and will be discussed later in this chapter.

Since P2P-Next focuses on video streaming, low playback latency is one of its most important requirements. Playback latency is fundamentally the sum of peer discovery delay plus buffering time since these two tasks are not parallelizable.

[1] `http://p2p-next.org` (September 2011)

These requirements added a practical motivation to our already established research motivation. The challenge is to develop a fully-decentralized peer discovery system which scales to millions of nodes and consistently achieves sub-second lookup latencies.

2.2 Distributed Hash Tables

As the name suggests, a distributed hash table (DHT) provides the functionality of a hash table. That is, it makes possible to *store* a *value* related to a given *key*. Conversely, it offers a *retrieve* operation which will retrieve the *value* previously stored under the *key*, if any.

Analogously to the traditional on-memory hash table, a DHT has a globally known hash function which maps keys into locations. The critical difference is that locations, in the case of the DHT, are not memory locations but node addresses.

Hash tables are known to perform store and retrieve operations in $O(1)$ because the calculation of the memory address from a given key does not depend on the size of the table, assuming constant access to any memory location, in average. DHT operations do depend on the number of nodes in the overlay, as we will see below, but the performance degradation is weak enough, making DHT overlays scalable.

In a DHT, each participating node is identified by a identifier called *nodeID*. Both node IDs and keys —keys are also known as *objectIDs*— are binary strings of the same size, usually derived from a hash function such as MD5, SHA-1 or SHA-256. In addition, each DHT design defines a globally known *distance function* which calculates the distance between two identifiers.

Each node in the DHT overlay knows the contact information of several other nodes. A node's contact information is the node's address in the underlying network (e.g., IP address and UDP port number).

In the extreme case where every node is aware of the location of every single node in the DHT overlay, a node would find the closest nodeID to the given key by calculating the distance between each nodeID and the key. Given that the node can sort the list of nodes by their nodeIDs, it can find the closest nodeID to the *target key* by simply performing a bisect search. This search operation is performed in $O(log(n))$ steps with n being the number of nodes in the DHT overlay. The result of this local search will be a node —or set of nodes— handling *store* and *retrieve* messages for that particular key.

It is easy to understand that in the case just described, nodes with millions of connections will be overwhelmed just keeping track of nodes joining and leaving the DHT overlay. Put in an other way, the load for each node would grow linearly with the number of nodes in the overlay, posing a serious thread to the scalability of such system.

In practice, nodes are not directly connected to every other node in the overlay. Instead, each node is connected to a small group of nodes (usually $O(log(n))$ nodes). This point can be easily illustrated by studying, for instance, a DHT overlay where

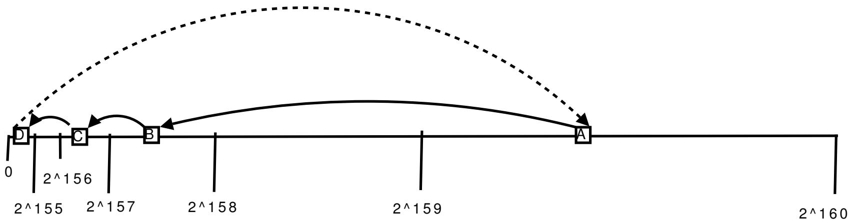

Figure 2.1: **Recursive routing.** Node A performs a lookup by sending a query (full line) to node B. Nodes relay the query to other nodes closer to the target. Node D, the closest node to the target, sends a response (dashed line) containing a value to node A. (Another valid option, not shown in the figure, is to send the response via the established routing path.)

each node is connected to $5 * log_2(n)$ nodes. When the number of nodes in the overlay is $1,000$ ($\sim 2^{10}$), each node is connected to around 50 nodes, which represent 5% of the total. For $1,000,000$ nodes ($\sim 2^{20}$), the number of connections is just 100 (0.01% of the total). It is clear that this model scales much better than a full-mesh network where every node wold be connected to every other node on the overlay.

These connections, strategically chosen, form the node's *routing table*. They are, in fact, used to route *store* and *retrieve* commands from any node to the —usually a small set of— closest nodes to the key. This routing operation is known as *lookup*.

A lookup is analogous to a bisect search where the search space is halved in each step. Therefore, lookups take $O(log(n))$ steps (or *hops*)to complete.

Figure 2.1 shows an example how node A can reach the closest node to a key k in $O(log(n))$ hops. Notice that the distance range's upper bound is, at least, halved in each hop. The distance between node A's nodeID and K —denoted as $d(A, K)$— is within $[2^{159}, 2^{160})$. The closest node to K in the whole DHT is node D —$d(D, K) \in [0, 2^{155})$. Node A is not directly connected to node D but is connected to a set of nodes whose distance to K is within $[0, 2^{159})$, among them node B whose distance to K is within $[2^{157}, 2^{158})$. Likewise, node B is connected to node C —$d(C, K) \in [2^{156}, 2^{157})$— which is connected to node D.

Lookups can be recursive or iterative. In *recursive routing*, whenever a node receives a query, the node selects the node —or a small set of nodes— closest to the key and forwards the message to it/them. If the node receiving the query is the closest node to the key, it performs the required operation and sends a response to the node performing the lookup. Figure 2.1 illustrates the example given above using recursive routing.

In the case of *iterative routing*, nodes do not forward messages. Instead, each node receiving a query selects the closest nodes to the key from its routing table and replies with a message containing <nodeID, address> for each of them. Once the closest node(s) to the key has been located, the operation is performed. In this

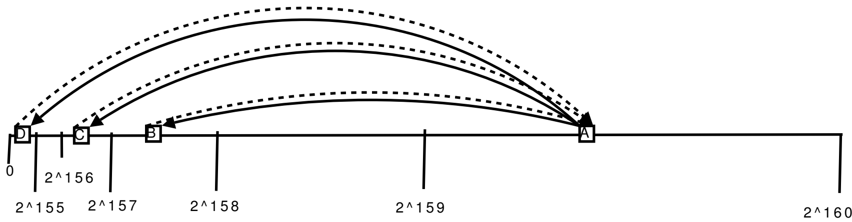

Figure 2.2: **Iterative routing.** Node A performs a lookup by sending a query (full line) to node B. Nodes respond to node A (dashed lines) with a list of nodes closer to the target. Node A keeps querying ever closer nodes until it finds the closest node to the key (node D in the figure).

case, nodes just provide routing information and the node performing the lookup is the only active actor in the whole lookup. Figure 2.2 shows how node A performs a lookup in an DHT overlay using iterative routing.

Notice that in both figures the horizontal line does not represent the identifier space in absolute terms, but the distance between each node to the target key. The distance between two identifiers is determined by the metric used by each DHT design. For instance, Kademlia uses $nodeID \oplus target$, while Chord uses $abs(nodeID - target)$ as distance function.

It is also worth noticing that both figures clearly illustrate how every hop reduces the distance to the key in, at least, half. As discussed above, this property guarantees lookups to terminate in $O(log(n))$ hops.

2.2.1 Kademlia: an XOR-metric Iterative DHT

In Kademlia [21], each node and object are assigned a unique identifier from the 160-bit key space, these identifiers are respectively called *nodeID* and *objectID*. Pairs of (*objectID*, *value*) are stored on nodes whose *nodeID* are closest to the *objectID*, where closeness is determined by performing an XOR bit-wise operation.

A node's routing table is organized in *buckets*, where each bucket contains up to k contacts sharing some common prefix with the routing table's owner. Each contact in the bucket is represented by the triple *(nodeID, IP address, UDP port)*.

The symmetric property of the XOR metric provides high flexibility and opportunistic routing table maintenance. These two benefits are briefly described here and will be exploited later (see Chapter 9) to improve the quality of routing tables and improve lookup performance.

Routing tables are flexible because a node can select which k nodes to add to each of its buckets from a potentially large amount of candidate nodes. For instance, consider a node whose nodeID starts with 100100. One of its buckets covers half of the identifier space (i.e. all nodes whose nodeID starts with 0), meaning that all

these nodes are valid candidates to add to this bucket.

Given this flexibility, it is possible to reduce maintenance costs by opportunistically finding and refreshing bucket entries. Whenever a message is received, the receiver can check which bucket the sender belongs to. If that bucket is not full, the receiver can consider adding the sender to the bucket. Although this has the benefit of filling buckets without additional messages, it can also create connectivity issues as we discuss in Chapter 8. Additionally, bucket entries can be opportunistically refreshed, reducing maintenance costs [21].

Kademlia's lookups are iterative. For each lookup, the node performing the lookup controls the lookup process as described above. This approach has been criticized because iterative routing can potentially add extra latency. For instance, when Crosby and Wallach [31] studied Kademlia's performance on large-scale DHT overlays (Mainline DHT and Azureus DHT), they hypothesized that one of the main reasons for such poor performance was due to Kademlia's iterative routing which adds a round-trip-time delay for each hop. They argued that a recursive approach would yield lower latencies because there is only one-way-trip-time delay per hop since nodes relay each others' lookup queries.

Iterative routing does have a key advantage over recursive routing. The fact that the node performing the lookup controls the lookup process means that local modifications on that node's lookup algorithm/configuration can have a significant effect on the node's lookup performance.

Not less important it is the fact that iterative routing allows us to study the whole lookup process by looking at a single node's traffic. We will exploit this opportunity to profile nodes' behavior and compare properties (such as performance and cost) different implementations of Mainline DHT nodes in Chapter 9.

2.3 Mainline DHT

The Mainline DHT overlay is an implementation of Kademlia. The most remarkable characteristic is the fact that values are not static but a dynamic list. This list contains the contact information <IP, port> of the peers participating in the swarm.

The process works as follows. A user produces a piece of content. She generates a *.torrent* file whose unique identifier (called *info hash* which identifies the content. When the user opens the .torrent file on her DHT-enabled BitTorrent client, the application will automatically announce itself to the DHT. That is, it will perform a lookup and store (on the nodes closest to the *info hash* key) its <IP, port>, where it offers the content via the BitTorrent protocol.

The elements in these lists expire after a period of time. This means that the peer needs to periodically announce itself to make sure it can be found by others.

A user interested in downloading this piece of content will obtain the content's *info hash*[2] and open it on his BitTorrent client. The software will then use MDHT

[2]There are several ways to distribute *info hashes*, all of them work outside the Mainline DHT overlay and hence out of the scope of this thesis.

to (1) find peers participating in the swarm and (2) announce itself so other peers can establish connections. The lookup is performed until the closest nodes to the info hash are identified. During the process, the nodes which have a list of peers associated to the *info hash*, will return it. With this information, the BitTorrent client establishes connections to some peers and starts downloading data. Once the lookup is done, the node will announce its peer's <IP, port> on the closest nodes. Those nodes will add the peer to the list of peers associated to the *info hash*.

Mainline DHT is, with over 5 million nodes [33], the largest DHT overlay ever deployed on the open Internet. Therefore, it offers an excellent opportunity for researchers to analyze a large-scale DHT overlay in the wild.

2.4 Related Work on Improving DHT Performance

Li et al. [34] simulated several DHTs under intensive churn and lookup workloads, comparing the effects of different design properties and parameter values on performance and cost. The study revealed that, under intensive churn, Kademlia's parallel lookups reduce the effect of timeouts compared to other DHT designs studied. In their simulations, Kademlia achieved a median lookup latency of 450 ms with their best parameter settings.

Kaune et al. [35] proposed *proximity neighbour selection (PNS)*, a mechanism to introduce a bias towards geographically close nodes in routing tables. Although their goal was to reduce inter-ISP Kademlia traffic, they observed that PNS also reduced lookup latency in their simulations from 800 to 250 ms.

Other non-Kademlia-based systems have been studied. Rhea et al. [25] showed that an overlay deployed on 300 PlanetLab hosts can achieve low lookup latencies (median under 200 ms and 99$^{\text{th}}$ percentile under 400 ms). Dabek et al. [36] achieved median lookup latencies between 100–300 ms on an overlay with 180 test-bed hosts.

Crosby and Wallach [31] measured lookup performance in two Kademlia-based large-scale overlays on the Internet, reporting a median lookup latency of around *one minute* in Mainline DHT and *two minutes* in Azureus DHT. They argue that one of the causes of such poor performance is the existence of dead nodes (non-responding nodes) in routing tables combined with very long timeouts. Falkner et al. [32] reduced ADHT's median lookup latency from 127 to 13 seconds by increasing the lookup cost three-fold.

Stutzbach and Rejaie [27] modified eMule's implementation of KAD to increase lookup parallelism. Their experiments revealed that lookup cost increased considerably while lookup latency improved only slightly. Their best median lookup latency was around 2 seconds.

Steiner et al. [28] also tried to improve lookup performance by modifying eMule's lookup parameters. Although they discovered that eMule's software architecture limited their modifications' impact, they achieved median lookup latencies of 1.5 seconds on the KAD overlay.

Chapter 3

Problem Definition

To our knowledge, Mainline DHT is the largest DHT overlay on the Internet. Its performance is, however, very poor compared to simulated and small-scale DHT overlays. In our attempt to improve Mainline DHT's performance, we encountered different research questions which are defined below.

Although these problems are defined in a context where the main goal is to improve performance, their definitions are not restricted to this specific goal. Instead, these problems are some generic problems of adapting an overlay to: (1) the underlying infrastructure (Internet in our case) and (2) the requirements defined by the application using the overlay (peer discovery for P2P video-streaming).

Others may use the knowledge and the open-source tools presented in this thesis to design, deploy, and evaluate their own system, whichever their application requirements and underlying infrastructure may be.

- **Lookup Performance**

  In terms of lookup performance, our goal is to achieve sub-second lookups in the Mainline DHT overlay. We not only aspire to a low-latency median lookup latency with a long tail of high-latency lookups —as others have reported in previous work on large-scale DHT, but truly sub-second results where only a minimal fraction of the lookups (less than one per cent) take over one second.

- **Underlying connectivity**

  It is well known that connectivity on the Internet is far from ideal. Nevertheless, many distributed systems have been designed on the assumption that underlying connectivity is reciprocal and transitive, although many of these designs do not explicitly state these assumptions. The only connectivity challenge commonly addressed is churn (caused by joins, leaves, and failures). Characterizing common connectivity artifacts on the Internet helps to understand the size of the problem, the potential effects of the underlying connectivity artifacts on the overlay's performance, and to devise mechanisms to adapt the overlay to the underlying network environment.

- **Backward compatibility**

  Deploying an overlay on the Internet is hard and once it has reached critical mass it becomes *the* standard. It then becomes a great hurdle for any new backward-incompatible replacement to be widely deployed. This point is well illustrated by the transition of IP from version 4 to 6.

  In this work, we propose, implement, and deploy backward compatible modifications to take advantage of an existing widely-deployed overlay. The advantages are clear and include: (1) the possibility of studying and testing our modifications on a real-world large-scale overlay and (2) the potential impact of our research on production systems. This approach also introduces challenges: (1) the modifications cannot be applied globally but locally, limiting our options; (2) any global modification must provide large benefits and a transition plan that allows the overlay to evolve one node at a time.

- **Evaluation tools**

  We need to be able to quantify properties such as performance and cost to be able to evaluate the effects of modifications in the nodes. Trade-offs need to be clearly presented so users and developers can determine which configurations fulfill their requirements.

  Evaluation tools are necessary to measure the impact of different modifications and explore these trade-offs.

- **Scalability and Locality**

  By using uniformly distributed keys and nodeIDs, nodes are assigned a similar amount of keys to be responsible for. In the case where all keys have associated a similar load, all nodes would handle a similar amount of load. In Mainline DHT, however, keys are associated to BitTorrent swarms, whose popularity wildly vary. Furthermore, the load associated to a key increases linearly with the popularity of the swarm. Mainline DHT does not have any load-balancing mechanism to address this problem.

  A peer discovery mechanism would also benefit from locality-aware mechanism which return results close to the requester's location. These mechanisms would decrease inter-ISP traffic, potentially benefiting users and ISPs.

Chapter 4

Thesis Contribution

4.1 List of Publications

- R. Jimenez and B. Knutsson
  "CTracker: a Distributed BitTorrent Tracker Based on Chimera"
  In Proc. eChallenges 2008, vol. 2, pp. 941-947
  Stockholm, Sweden, Oct. 2008.

- R. Jimenez, F. Osmani, and B. Knutsson
  "Connectivity Properties of Mainline BitTorrent DHT Nodes"
  9th IEEE International Conference on Peer-to-Peer Computing 2009
  Seattle, Washington, USA, Sept. 2009.

- R. Jimenez, F. Osmani, and B. Knutsson
  "Sub-Second Lookups on a Large-Scale Kademlia-Based Overlay"
  11th IEEE International Conference on Peer-to-Peer Computing 2011
  Kyoto, Japan, Aug. 2011.

Publications of the same author not included in this work

- R. Jimenez, L.-E. Eriksson, and B. Knutsson
  "P2P-Next: Technical and Legal Challenges"
  In The Sixth Swedish National Computer Networking Workshop and Ninth Scandinavian Workshop on Wireless Adhoc Networks (poster)
  Uppsala, Sweden, May 2009.

- V. Grishchenko, F. Osmani, R. Jimenez, J. Pouwelse, and H. Sips
  "On the Design of a Practical Information-Centric Transport"
  PDS Technical Report PDS-2011-006, TUDelft
  Delft, the Netherlands, March 2011.

4.2 Scalability and Locality-Awareness

The work on scalability and locality on a DHT-based peer discovery system has been published in a conference paper [1] and appears in Chapter 7.

In this study, our experiments on a popular BitTorrent tracker show that there is ample room for improvement in locality-awareness, using round-trip-time as locality metric.

We propose a modified design of Tapestry [20] which addresses both scalability issues caused by popular keys and provides locality-aware features, with the intention to reduce inter-ISP BitTorrent traffic. These modifications are planned to be integrated into Mainline DHT in future work.

The mechanism presented in this work dynamically adapts the number of nodes responsible for a given key according to the key's popularity, thus mitigating the scalability issues. By introducing a bias towards low-latency neighbor selection, we also obtain locality-aware properties for popular keys.

4.3 Connectivity Properties

The work on connectivity properties has been published in a conference paper [2] and appears in Chapter 8.

In this work, we defined three connectivity properties that Kademlia nodes must have in order to be able to properly route lookups and store values: *reciprocity*, *transitivity*, and *persistence*. We give an account of how the lack of any one of them negatively affects the ability of nodes to carry out routing and storing operations. Furthermore, these *impaired* nodes are not merely failing to contribute resources to the network. When these nodes try to contribute their resources to the system, their connectivity issues cause routing failures, disrupting other nodes' lookups.

In our survey of over 3.6 million nodes in Mainline DHT, almost two-thirds of them could be considered as *impaired*. That leaves an environment where one-third of the nodes do useful work while two-thirds are useless at best, harmful at worst.

Finally, mechanisms are proposed to identify and discard nodes with connectivity issues to improve the quality of routing tables, thus reducing the routing failure rate. The integration of these mechanisms have shown to substantially improve lookup performance.

4.4 Sub-Second Lookups on a Multimillion-Node DHT Overlay

The work on achieving sub-second lookups on a multimillion-node overlay has been published in a conference paper [40] and appears in Chapter 9.

In our work, we survey the literature on DHT lookup performance finding no reports of sub-second lookups on large-scale (over one million nodes) DHT overlays.

On the other hand, sub-seconds results are common when performance is measured on simulators and small-scale overlays.

We argue that this discordance between laboratory and *real-world* performance is due to the non-ideal conditions of the underlying network, based on our previous analysis of connectivity artifacts on the Internet and their effect on Mainline DHT.

Our main contribution is to show that it is possible for a node participating in a multimillion-node Kademlia-based overlay to consistently perform sub-second lookups (median below 200 ms, 99th percentile below 600 ms). The modifications needed to achieve such performance are completely backward-compatible and can be incrementally integrated into the existing DHT overlay.

In our efforts to accomplish the goal of supporting latency-sensitive applications using a large-scale overlay, we also produced the following results: (1) a profiling toolkit that allows us to analyze DHT traffic without code instrumentation, (2) deployment and measurement of nodes whose routing table management and lookup algorithm were modified, and (3) the infrastructure necessary to deploy and evaluate these modifications.

4.5 Source Code

All source code used to produce these results is freely available, under an open-source license, for others to use and/or reproduce our experiments at:
`http://people.kth.se/~rauljc/lic/`

4.6 Individual Contribution

I am the main author of all the papers presented in this thesis and wrote most of the code. I led the work from the initial idea to writing, and presented them at the conferences. Instead of listing my individual contributions, I list the contributions of others:

In all the papers, Björn Knutsson discussed the ideas and co-edited the papers, contributing with very valuable guidance and comments.

In papers B and C, Flutra Osmani actively assisted in designing and running the experiments, although I wrote most of the necessary code and processed all the results. She discussed the ideas with me as they evolved and co-edited the papers.

Chapter 5

Discussion

The work presented in this thesis started with our focus on designing a fully-decentralized locality-aware tracking mechanism for BitTorrent. The motivation for such goal was two-fold: (1) reduce inter-ISP traffic by facilitating the discovery of peers within the same ISP and (2) mitigate the scalability issues caused by popular key —related to popular content— in the DHT.

Our proposal was based on a recursive DHT design called Tapestry [20] plus our own modifications to construct low-latency routing tables based on round trip times and a simple mechanism to avoid hotspots.

This proposal also considered lookup performance. In theory, a recursive DHT can yield lower latencies than an iterative one, since it avoids several round trips when the number of hops is large enough [31][1]. In addition, both of our modifications (low-latency neighbors and spread out of values in the DHT) would lower lookup latency as a side effect.

When we tried to implement and deploy our design, we realized that deploying a large-scale overlay would require not only hard work, but also collaboration with existing developing teams. We concluded that our potential impact on real-world systems would probably be greater if we introduced backward-compatible incremental modifications to an already widely deployed overlay.

We decided to study one of the large-scale DHT overlays on the open Internet. We chose Mainline DHT for two reasons: (1) MDHT is the largest DHT overlay on the open Internet and (2) Tribler [41], the BitTorrent client whose code-base is used by P2P-Next, already supported MDHT.

In our study of Mainline DHT, we analyzed previous measurements of its performance [31]. Mainline DHT's poor performance was not the exception, but the norm. All the other large-scale DHT overlays on the open Internet also perform poorly compared to performance measurements on simulators and small-scale Kademlia-based overlays.

[1] In practice, however, we have seen that iterative routing is very flexible and exploiting the possibility of parallel queries can yield low lookup latencies.

Some previous work identified connectivity artifacts as a cause for such poor performance [31, 37]. That led us to investigate what connectivity properties where assumed/required by DHTs (Kademlia in particular) and what kind of connectivity was actually available on the Internet (Mainline DHT nodes in particular).

Once these properties were defined and measured, we analyzed the effects of the observed connectivity artifacts on Kademlia, concluding that there is a clear mismatch between Kademlia's networking assumptions/requirements and the actual underlying connectivity. Then, we designed backward-compatible mechanisms to identify these connectivity artifacts and mitigate their impact on lookup performance.

Finally, we demonstrated the effectiveness of our mechanisms by measuring the performance of nodes implementing them. Our results definitively close the large performance gap between laboratory and Internet-wide multi-million DHT overlays.

We consider that the performance issue has been successfully addressed in thesis. Deciding on the specific trade-off details is now an engineering problem for BitTorrent developers. They are not alone, though. We offer them our experience and tools to experiment with different mechanisms and parameters so they can achieve the performance they require at a cost they can afford.

Chapter 6

Conclusion

This thesis has studied the Mainline DHT overlay, which with 5 to 9 million simultaneous nodes is the largest DHT overlay on the Internet. This DHT overlay is based on Kademlia and is used as peer discovery mechanism (or *tracker*) in BitTorrent.

The main result of this thesis is a dramatic lookup performance improvement on Mainline DHT. This improvement is the result of our backward-compatible modifications of routing table management and lookup algorithm configuration.

We have shown that our novel modifications on routing table management not only improve performance of the node implementing them, but also have the potential to improve performance globally. These modifications, combined with lookup configurations already studied in the literature, yield median lookup latencies far lower than previous measurements on large-scale DHT overlays.

While median lookup latency is important, consistent low-latency is critical for the viability of DHT-based latency-sensitive applications. We have also succeeded in this point, achieving sub-second latencies in well above 99% of the lookups.

These results challenge the idea that Internet-wide DHT overlays are not suitable for latency-critical applications due to their inherent poor performance. While there are still several important issues to address —some of them listed in Future Work, our work provides the tools to effectively mitigate the poor performance problem on large-scale Kademlia-based overlays.

In our quest to improve Mainline DHT performance we have studied important deployment details, in particular how underlying connectivity artifacts affect DHT overlays. Our analysis uncovered that connectivity artifacts are common on the underlying network (i.e. the Internet) used by the Mainline DHT overlay. Furthermore, these connectivity artifacts have the potential to pollute routing tables and degrade lookup performance. Since the original Kademlia design implicitly assumes a nearly-ideal underlying connectivity, these connectivity artifacts would explain not only Mainline DHT poor performance, but also all other large-scale Kademlia-based overlays documented in the literature.

We designed a series of modifications on the routing table management policy

and lookup algorithm based on our analysis of underlying connectivity and its effects on Kademlia-based DHT overlays. These modifications, as reported above, have proven to dramatically improve lookup performance.

Conscious of the importance of powerful and reliable tools on our long-term study of large-scale DHT overlays, we have built a rich open-source software repository. All necessary software to reproduce the results presented in this thesis are available on-line at `http://people.kth.se/~rauljc/lic/`.

Finally, we have also proposed a mechanism to both address scalability challenges and add locality-aware features to DHT-based peer discovery systems. Although this proposal is based on a non-Kademlia DHT design, we consider integrating these modifications into Mainline DHT as future work.

6.1 Future Work

6.1.1 Scalability and Locality

This thesis has addressed the scalability and locality issues, albeit solely from a theoretical point of view on a recursive DHT. Our intention is to use all the experience with Mainline DHT, gained in the process of improving lookup performance, to integrate scalability and locality improvements proposed in this thesis.

6.1.2 Privacy and Security

Although the Mainline DHT specification [39] states *"Node IDs are chosen at random from the same 160-bit space as BitTorrent infohashes"*, each Mainline DHT node is free to choose its own nodeID.

This policy has privacy and security implications. From a privacy point of view, it is trivial for an attacker to create a node whose nodeID is close to a *key of interest*. This node would became one of the trackers for that *content of interest*, collecting data about users participating in the exchange of that content.

From a security point of view, Mainline DHT does not have any mechanism to defend itself against a Sybil attack [42]. A Sybil attack can potentially restrict access to a given piece of content by locating a large amount of nodes close to the content's identifier and returning bogus lists of peers. Some work has been done in this direction. In a master thesis under my supervision, Ismael Saad Garcia [43] explored the effects of these kind of attacks on Mainline DHT.

These security issues are known in the BitTorrent community and one interesting tentative solution has been proposed[1]. In the research community, an excellent survey by Urdaneta et al. [44] covers the related work on DHT security.

[1] `http://www.rasterbar.com/products/libtorrent/dht_sec.html` (Oct. 2011)

Bibliography

[1] R. Jimenez and B. Knutsson, “CTracker: a Distributed BitTorrent Tracker Based on Chimera,” in *In Proc. eChallenges 2008*, vol. 2, pp. 941–947, Oct. 2008.

[2] R. Jimenez, F. Osmani, and B. Knutsson, “Connectivity properties of Mainline BitTorrent DHT nodes,” in *9th International Conference on Peer-to-Peer Computing 2009*, (Seattle, Washington, USA), 9 2009.

[3] R. Jimenez, L.-E. Eriksson, and B. Knutsson, “P2p-next: Technical and legal challenges,” in *The Sixth Swedish National Computer Networking Workshop and Ninth Scandinavian Workshop on Wireless Adhoc Networks*, (Uppsala, Sweden), May 2009.

[4] D. Hastorun, M. Jampani, G. Kakulapati, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: Amazon’s highly available key-value store,” in *In Proc. SOSP*, Citeseer, 2007.

[5] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Commun. ACM*, vol. 51, pp. 107–113, January 2008.

[6] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: distributed data-parallel programs from sequential building blocks,” *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 59–72, March 2007.

[7] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Leboisky, “Seti@home-massively distributed computing for seti,” *Computing in Science Engineering*, vol. 3, pp. 78 –83, jan/feb 2001.

[8] G. Kreitz and F. Niemela, “Spotify - Large Scale, Low Latency, P2P Music-on-Demand Streaming,” in *P2P’10*, 2010.

[9] S. Saroiu, K. Gummadi, and S. Gribble, “Measuring and analyzing the characteristics of napster and gnutella hosts,” *Multimedia systems*, vol. 9, no. 2, pp. 170–184, 2003.

[10] B. Cohen, “BitTorrent Enhancement Proposal 3 (BEP3): The BitTorrent Protocol Specification,” 2008.

[11] Y. Kulbak and D. Bickson, "The emule protocol specification," *eMule project*, 2009.

[12] I. Clarke, O. Sandberg, B. Wiley, and T. Hong, "Freenet: A distributed anonymous information storage and retrieval system," in *Designing Privacy Enhancing Technologies*, pp. 46–66, Springer, 2001.

[13] M. Ripeanu, "Peer-to-peer architecture case study: Gnutella network," in *First International Conference on Peer-to-Peer Computing*, pp. 99–100, IEEE, 2001.

[14] H. Zhang, A. Goel, and R. Govindan, "Using the small-world model to improve freenet performance," *Computer Networks*, vol. 46, no. 4, pp. 555 – 574, 2004.

[15] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker, "Making gnutella-like p2p systems scalable," in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '03, (New York, NY, USA), pp. 407–418, ACM, 2003.

[16] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica, "The impact of DHT routing geometry on resilience and proximity," in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 381–394, ACM, 2003.

[17] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, (New York, NY, USA), pp. 161–172, ACM, 2001.

[18] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[19] A. Rowstron and P. Druschel, "P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," *In: Middleware*, pp. 329–350, 2001.

[20] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz, "Tapestry: a resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 41–53, 2004.

[21] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the XOR metric," in *Proceedings of the 1st International Workshop on Peer-to Peer Systems (IPTPS02)*, pp. 53–65, 2002.

[22] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu, "Opendht: a public dht service and its uses," in *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, (New York, NY, USA), pp. 73–84, ACM, 2005.

[23] M. J. Freedman, “Experiences with coralcdn: a five-year operational view,” in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI’10, (Berkeley, CA, USA), pp. 7–7, USENIX Association, 2010.

[24] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, “Planetlab: an overlay testbed for broad-coverage services,” *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 3, pp. 3–12, 2003.

[25] S. Rhea, B. Chun, J. Kubiatowicz, and S. Shenker, “Fixing the embarrassing slowness of OpenDHT on PlanetLab,” in *Proc. of the Second USENIX Workshop on Real, Large Distributed Systems*, pp. 25–30, 2005.

[26] B. Cohen, “Incentives Build Robustness in BitTorrent,” in *Workshop on Economics of Peer-to-Peer Systems*, vol. 6, Berkeley, CA, USA, 2003.

[27] D. Stutzbach and R. Rejaie, “Improving Lookup Performance Over a Widely-Deployed DHT,” in *INFOCOM*, IEEE, 2006.

[28] M. Steiner, D. Carra, and E. W. Biersack, “Evaluating and improving the content access in KAD,” *Springer "Journal of Peer-to-Peer Networks and Applications", Vol 2*, 2009.

[29] M. Steiner, T. En-Najjary, and E. W. Biersack, “A global view of kad,” in *IMC ’07: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, (New York, NY, USA), pp. 117–122, ACM, 2007.

[30] D. Carra, M. Steiner, and P. Michiardi, “Adaptive load balancing in kad,” in *Peer-to-Peer Computing (P2P), 2011 IEEE International Conference on*, pp. 92 –101, 31 2011-sept. 2 2011.

[31] S. A. Crosby and D. S. Wallach, “An analysis of bittorrent’s two kademlia-based dhts,” 2007.

[32] J. Falkner, M. Piatek, J. P. John, A. Krishnamurthy, and T. Anderson, “Profiling a million user DHT,” in *IMC ’07: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, (New York, NY, USA), pp. 129–134, ACM, 2007.

[33] K. Junemann, P. Andelfinger, J. Dinger, and H. Hartenstein, “BitMON: A Tool for Automated Monitoring of the BitTorrent DHT,” in *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on*, pp. 1–2, IEEE, 2010.

[34] J. Li, J. Stribling, R. Morris, M. F. Kaashoek, and T. M. Gil, “A performance vs. cost framework for evaluating DHT design tradeoffs under churn,” in *INFOCOM*, pp. 225–236, 2005.

[35] S. Kaune, T. Lauinger, A. Kovacevic, and K. Pussep, "Embracing the peer next door: Proximity in kademlia," in *Eighth International Conference on Peer-to-Peer Computing (P2P'08)*, p. 343–350, 2008.

[36] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris, "Designing a dht for low latency and high throughput," in *IN PROCEEDINGS OF THE 1ST NSDI*, pp. 85–98, 2004.

[37] M. J. Freedman, K. Lakshminarayanan, S. Rhea, and I. Stoica, "Non-transitive connectivity and dhts," in *In Proc. of the 2nd Workshop on Real Large Distributed Systems*, 2005.

[38] M. Varvello and M. Steiner, "Traffic localization for dht-based bittorrent networks," in *NETWORKING 2011* (J. Domingo-Pascual, P. Manzoni, S. Palazzo, A. Pont, and C. Scoglio, eds.), vol. 6641 of *Lecture Notes in Computer Science*, pp. 40–53, Springer Berlin / Heidelberg, 2011.

[39] A. Loewenstern, "BitTorrent Enhancement Proposal 5 (BEP5): DHT Protocol," 2008.

[40] R. Jimenez, F. Osmani, and B. Knutsson, "Sub-Second Lookups on a Large-Scale Kademlia-Based Overlay," in *11th International Conference on Peer-to-Peer Computing 2011*, (Kyoto, Japan), 8 2011.

[41] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. J. Epema, M. Reinders, M. R. van Steen, and H. J. Sips, "Tribler: a social-based peer-to-peer system: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 20, no. 2, pp. 127–138, 2008.

[42] J. Douceur, "The sybil attack," *Peer-to-peer Systems*, pp. 251–260, 2002.

[43] I. S. Garcia, "Exploring Mainline DHT: an experimental approach," *KTH Master Thesis*, November 2010.

[44] G. Urdaneta, G. Pierre, and M. van Steen, "A survey of DHT security techniques," *ACM Computing Surveys*, vol. 43, Jan. 2011. `http://www.globule.org/publi/SDST_acmcs2009.html`.

Part II

Research Papers

Paper A

Chapter 7

CTracker: a Distributed BitTorrent Tracker Based on Chimera

R. Jimenez, B. Knutsson

In *eChallenges 2008*, vol.2, pp. 941–947, IOS Press.
Oct. 22–24, 2008, Stockholm, Sweden

© 2008 The authors. Reprinted with permission.

CTracker: a Distributed BitTorrent Tracker Based on Chimera

Raúl JIMÉNEZ, Björn KNUTSSON
Kungliga Tekniska högskolan, Isafjordsgatan 39, Stockholm, 164 40, Sweden
Tel: +46 8 790 42 85, Fax: +46 8 751 17 93, Email: rauljc@kth.se; bkn@kth.se

Abstract: There are three major open issues in the BitTorrent peer discovery system, which are not solved by any of the currently deployed solutions. These issues seriously threaten BitTorrent's scalability, especially when considering that mainstream content distributors could start using BitTorrent for distributing content to millions of users simultaneously in the near future.
In this paper these issues are addressed by proposing a topology-aware distributed tracking system as a replacement for both centralized and Kademlia-based trackers.
An experiment measuring most popular open BitTorrent trackers is also presented. It shows that centralized trackers are not topology aware. We conclude that an ideal topology-aware tracker would return peers whose latency to the requester peer is significantly lower than of a centralized tracker.

1. Introduction

The BitTorrent protocol [1] distributes digital content using the resources every participant offers. Every participant is called a peer and a swarm is a set of peers participating in the distribution, i.e., downloading and uploading of a given content.

Nowadays the most popular swarms on the most popular BitTorrent public trackers hold a few tens of thousand peers. BitTorrent usage is continuously increasing and with the participation of legal content distributors we can expect the usage to skyrocket. When content providers start distributing popular TV shows and movies through BitTorrent we should not be surprised to have several millions of peers participating simultaneously in a single swarm.

Big players are already moving towards on-line content distribution by using different peer-to-peer and hybrid systems, for instance, BBC with its successful iPlayer [2]. Furthermore, among others, BBC and the European Broadcaster Union participate in the P2P-Next project [3]. P2P-Next is a Seventh Research Framework Programme project, which aims to become the standard for on-line content distribution using the BitTorrent framework.

The BitTorrent tracker is a key component of the BitTorrent framework. A tracker is set up in order to track the participants within a swarm. Every peer willing to join the swarm contacts the tracker and requests a list of peers participating in the swarm. Then this peer will contact the peers in the list in order to download/upload content from/to them.

Unfortunately, there are three major open issues that threaten the reliability of the tracking system. (1) The tracker is a single point of failure. When a tracker fails the current members of the swarm are not affected but no other peer will be able to join. (2) The tracker faces a scalability issue since it is only able to handle a finite number of peers based on the processing power and bandwidth available. (3) The third issue is locality, and it is more subtle: when you contact the tracker, you will receive a random subset of the available peers. This means that while a local peer may exist, you may only be notified of peers on

other continents, resulting in both higher global bandwidth consumption and lower download speed.

The single point of failure issue has been addressed by distributing the tracking tasks among the peers. There are two implementations both based on a DHT (Distributed Hash Table) called Kademlia [4]. Several problems have, however, been found on both of them [5] and there is no visible effort towards fixing them because they are not considered critical but a backup mechanism should the tracker fail.

The scalability issue also affects the distributed trackers because the small set of nodes that are responsible for tracking a specific swarm (8 or 20 nodes in the current implementations) will receive every query. Kademlia partially distributes this responsibility by caching part of the address list on the nearby nodes. This caching feature, however, is not good enough. Although the caching nodes help replying requests, the core nodes must still keep track of every single peer in the swarm.

The locality issue is a consequence of the equality of the peers. From the tracker's point of view there is no discernible difference among peers, therefore it is not able to return a list consisting of the “best” peers, but rather just a random set of peers. In order to improve locality, the tracker could use different heuristics such as geolocation services but that would imply an extra overload. Probably that is the reason why trackers lack this feature.

This paper proposes a system that addresses the three issues described in this section.

2. Objectives

The main objective of this paper is to present a design for a topology-aware scalable distributed tracker based on Chimera, which addresses the issues explained in the introduction. In addition, we will outline some additional benefits of our proposed design.

We also show, through our simple experiment, that there is ample room for improvement. The difference between the ideal tracker and the current centralized tracker implementations is large enough to justify the research on this topic and the replacement of the current tracking system.

3. System Overview

We have undertaken a study of the behavior of the BitTorrent protocol and extensions, and the currently available DHT technologies. Based on the results, we designed and implemented a prototype and compared its behavior with the existing BitTorrent implementations [6].

In this paper a different approach is suggested. Instead of designing, implementing and deploying a completely new protocol we propose to replace just the DHT system in the current BitTorrent framework. We consider that, by being BitTorrent backwards compatible, this DHT replacement can be implemented and deployed more easily, increasing drastically the probability of a large-scale deployment.

Furthermore, we are considering, together with the Tribler research team, to integrate Chimera's key properties into the existing Kademlia-based DHT. If this is possible, the new solution would be fully backwards compatible with Mainline DHT clients. This task, however, is out of the scope of this paper and regarded as future work.

3.1 Distributed Tracker within the BitTorrent Framework

BitTorrent applications using a distributed tracker have two components: (1) a peer which uses the BitTorrent protocol to download/upload data from/to other peers and (2) a node that is a member of the DHT and performs the distributed tracker's tasks.

As stated in the introduction the existing implementations of distributed trackers fail to address the scalability and locality issues. We consider that a topology aware framework offers us the properties needed to address these issues. This framework would allow us to

spread small lists of topologically close peers among the nodes; contrary to assigning the task of tracking the whole swarm to a small set of nodes (one node plus a few replicas).

There is a framework called Chimera whose properties fulfill the requirements for such system.

3.2 Chimera's Routing Algorithm

In this subsection a brief description of Chimera's behavior is given. Further information about Chimera is located in the Related Work section.

Chimera [7] is a topology-aware DHT overlay. It routes each message through a number of nodes until it reaches the destination node. Every node in the path processes the message and routes it to another node whose identifier is closer –i.e, more prefix matching bits– to the destination identifier. A node is a destination of a message when there is no node whose identifier is closer in the identifier space –node and destination identifiers might match but that case is very rare.

When routing a message, a node forwards it to the topologically closest node among the candidates. This behavior makes Chimera topology aware, especially during the first hops into the DHT; contrary to the current BitTorrent DHT's behavior, where hops are randomly long all the way to the destination.

Furthermore, intermediate nodes can cache and retrieve results, a key property used by our design, which will be explained in depth later.

3.3 Integrating Chimera as Distributed BitTorrent Tracker

Nodes can send two kind of messages: announce and find_peers. Every message is routed according to a modified version of Chimera's routing algorithm that is explained along this section.

An announce message contains a [IP, port number] pair and its destination is a swarm identifier. This message announces that this peer is participating in a swarm and where it can be contacted by other peers. Every node in the path stores the [peer, swarm] pair and routes the message. There is no reply for this message.

A find_peers message is addressed to a swarm identifier and it is created by a node looking for peers participating in the swarm. Every node in the path checks whether it has information about the swarm. If there is a list of peers for that swarm, that list is sent to the requester. Otherwise the message is forwarded to the next node.

So far the scalability related to the centralized tracker and locality issues have been addressed, allowing intermediate nodes to return small lists of topologically-close peers. This is still not good enough, however, since the destination of a very popular swarm –say 10 million peers– will receive every single announce message and keep track of every peer.

Solving this issue is the main contribution of this paper and it justifies replacing the current DHT used in the BitTorrent framework.

3.4 Scalability Improvement over the Current Distributed Trackers

Chimera's routing algorithm can be modified to forward only a limited number of announce messages. In this way, destination nodes tracking a few tens of peers will keep track of every peer in the swarm but when the swarm reaches 10 millions of peers this node will only track a limited number of peers (the topologically closest peers).

In the modified routing algorithm there are two new parameters m and n where $n \geq m$. The parameter m is the maximum number of announce messages to be forwarded per swarm and n is the maximum number of peers stored in a swarm list. The swarm list is ordered by the distance –network latency– from the node to the peers stored in the list. These parameters can be calculated independently by every node and might be dynamic depending the node's configuration and workload. For instance, a powerful node which

wants to store every announcement received might set n to infinite, however, it must be more careful setting m in order to keep the DHT bandwidth overhead low.

Every node in the path of an announce message measures the latency to the new peer and tries to add it to the swarm list. If the list length is already n and the new peer is not closer than any other in the list then the message is dropped. If the list length is between *n* and m, the new peer will be added to the list, but the message will only be forwarded if the new element is inserted among the m lowest latency peers. Lastly, if the list is shorter than m elements, the message is forwarded following the original Chimera routing algorithm.

One may think that the fact a high latency node (e.g, satellite connections) can be isolated by dropping its announce messages is a design flaw. If this node happens to join a busy swarm and the next node in the Chimera overlay has already a list with n elements, the message will be dropped and there will be no reference to its participation in the whole DHT. Unfortunately for this node, that is exactly what is desired; close nodes are easy to find and the far-away ones are not. This node can, however, always send find_peers messages and discover other peers, therefore there is no risk of total isolation. In a sense, it would have the same effect as a peer behind a NAT or firewall, where the peer can establish connections to others but cannot be contacted by other peers.

3.5 Additional Benefits

Not only will this system improve tracker's scalability, but it can also decrease costs for ISPs. Being able to find topologically close peers, peers can easily discover other peers within the same ISP, thus reducing inter-ISP traffic. Furthermore, ISPs could offer users incentives to decrease inter-ISP traffic even further (e.g., by increasing link speed in connections within the ISP's network and/or setting up an easy to find peer offering cached content).

This is not a minor benefit, since BitTorrent traffic represents an important fraction of the total Internet traffic [8] and some ISPs are trying to control this by caching, throttling or banning BitTorrent traffic, in order to reduce costs and impact on other traffic [9-10].

4. Centralized Tracker Versus Topology-Aware Decentralized Tracker

In this section, the results from a small-scale experiment show how topology (un)aware the most popular BitTorrent centralized trackers are. Then, these results are compared to an ideal topology-aware decentralized tracker.

The BitTorrent specifications [11] do not specify how many peers a tracker should return as a response to a peer's query, nor how these peers should be selected. It is believed that most of the tracker implementations return a list of random peers, i.e., trackers are not topology aware.

In our experiment, the most popular torrent files in Mininova.org were downloaded. Since mininova offers torrent files tracked by different trackers, several tracker implementations are analyzed at once. The results show, however, that there are no discernible differences among different trackers regarding topology awareness.

A total of 79 swarms were analyzed. Every 5 minutes a request was sent to every tracker, obtaining 79 lists of peers. Then, the latency to these peers was measured by using tcptraceroute to every peer in the list. This process was repeated 5 times.

One of the most interesting findings is that most of the peers were not reachable on the port announced to the tracker. The suggested explanations are: peers no longer on-line, NATs, and firewalls; and has been reported by other experiments on BitTorrent [12]. In total, 11578 reachable peers were measured; around 150 peers per swarm (i.e., 30 reachable peers per request on average.

In Figure 1 the average latency to the peers is plotted. For every swarm there are five points and two curves, where five points represent the average latency to the peers in the

list returned to each request. One curve represents the average latency to every peer measured within the swarm while the other shows the average latency to the *x* lowest latency peers in the swarm.

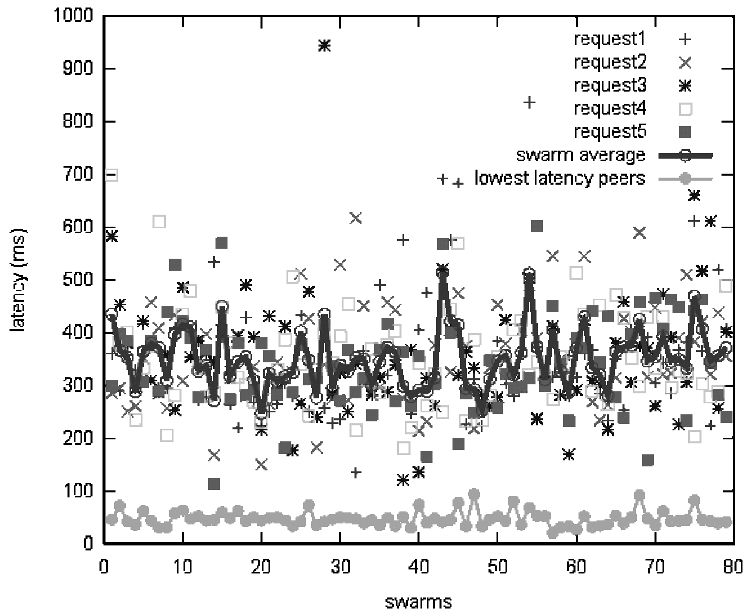

Figure 1: Latency measurements to peers participating in popular BitTorrent swarms

This last curve represents the ideal list of peers that a topology aware tracker should return and it is calculated as follows. The tracker returns 5 lists of torrents as response to our 5 requests. We check the reachability of peers in every list and calculate the average number of reachable peers per request which will be called *x*. If the tracker were ideally topology-aware it should have returned just the *x* lowest latency peers whose average is plotted in the figure forming the "lowest latency peers" curve.

The figure shows that the average latency for different requests is as random as we can expect, when assuming that trackers return lists of random peers. It also shows that the difference between the average latency to every peer returned (swarm average) and to the ideal list (lowest latency peers) is between 5 and 10 times. While the former backs our initial assumption, the latter shows a large room for improvement in BitTorrent trackers.

Our measurements so far have confirmed our hypothesis, but we are continuously working to study larger swarms, and we will also start monitoring swarms from multiple vantage points.

5. Related Work

In this section background information about DHT and Chimera is given.

5.1 Distributed Hash Tables

Several structured lookup protocols [13] have been studied in order to choose one that offers the characteristics this system needs. Chord [14] is one of the most well-known DHT. Actually, Kademlia [4] is a Chord derivation used in BitTorrent. Although these protocols provide a distributed lookup system, they do not offer topology awareness.

On the other hand, Tapestry [15] is a structured lookup protocol that provides this characteristic. Moreover, its implementation in C –called Chimera– is flexible enough to be

adapted to our needs. In this project, Chimera was chosen as lookup overlay, whose description will be explained next.

5.2 Chimera

As an implementation of Tapestry, Chimera routes messages from one node to a destination's root. In Chimera, each node has a unique identifier. Each node has several routing tables, with references to its nearest neighbors within a level.

A link belongs to a level, depending on the length of the shared prefix of the identifiers of the two nodes involved. For instance, let identifiers in hexadecimal and 4-bit levels, node 6E83 has links of level 4 to nodes whose identifier are 6E8*, level 3 to nodes 6E** and so on. In fact, this is similar to IP routing.

A message from one node to another is routed choosing the highest level link in each step. Since each step routes the message through a greater level, the maximum number of hops is $\log_{\beta}(N)$, where the identifiers are expressed in base β and N is the length of the identifier. Moreover, since each node routes the messages through its nearest neighbor in that level, the paths are deterministic and topologically aware. At the last hop, the message's and the node's identifiers match, then the message is delivered.

Each object –torrent identifier– has a unique identifier as well. When any node wants to perform an operation over an object (publish, unpublish, lookup, etc.), the message is routed to the block's identifier. Since most likely there will not be a node matching it, the message will reach its destination's root. This is the node whose identifier is the closest to the block's one.

Then, a publish(objectID) is delivered to the objectID's root and this node stores all the references to objectID. In the path, each node which forwards the publish messages also stores the references. A lookup message will be routed in the same manner, however, at any hop it will reach a node which stores a list of references –list of peers participating in the swarm. This node can stop the lookup and return its list of references. This list will be shorter than the root's one and these references were likely published by the closest nodes to the requester.

The main difference between Chimera and Tapestry is that Tapestry reaches the node that actually published an object, while Chimera only routes the messages. Because of the aforementioned property it was possible to use Chimera in this design.

6. Conclusions

In this paper a design of a topology-aware distributed BitTorrent tracker has been presented. This design addresses three key open issues in the current BitTorrent tracker system. We explained how the scalability of the whole system is improved drastically by removing the existence of hot spots in the DHT. This is a desirable property nowadays but it will be absolutely necessary when mainstream content providers offer their content on the BitTorrent framework in the near future.

ISPs will play a key role in P2P content distribution. This design provides mechanisms to reduce inter-ISP traffic, improving user experience (lower lookup latency and increased download speed), without increasing dramatically the ISP's costs.

Our experiment has shown how the most popular open BitTorrent trackers behave and how far their results are from an ideal topology aware system. This backs our initial hypothesis that there is a large room for improvement and encourages us to implement the described system in order to measure its performance.

Future work includes modifying Tribler [16], a BitTorrent-based content distribution framework developed by a research team at Delft University, integrating this design, and comparing performance against the current unmodified version.

Acknowledgment

The research leading to these results has received funding from the Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 21617.

References

[1] B. Cohen. Incentives build robustness in BitTorrent. In Proceedings of the First Workshop on the Economics of Peer-to-Peer Systems, Berkeley, CA, June 2003.
[2] BBC. iPlayer. http://www.bbc.co.uk/iplayer/ (April 2008)
[3] P2P-Next. http://www.p2p-next.org/ (April 2008)
[4] P. Maymounkov and D. Mazieres. Kademlia: A peerto -peer information system based on the xor metric. In Proceedings of IPTPS02, Cambridge, USA, March 2002. http://www.cs.rice.edu/Conferences/IPTPS02/.
[5] Scott A Crosby and Dan S Wallach An Analysis of BitTorrent's Two Kademlia-Based DHTs Technical Report TR-07-04, Department of Computer Science, Rice University, June 2007.
[6] R. Jiménez. Ant: A Distributed Data Storage And Delivery System Aware of the Underlaying Topology. Master Thesis. KTH, Stockholm, Sweden. August 2006.
[7] CURRENT Lab, U. C. Santa Barbara. Chimera Project. http://current.cs.ucsb.edu/projects/chimera/ (April 2008)
[8] A. Parker. The true picture of peer-to-peer filesharing, 2004. http://www.cachelogic.com/. (April 2008)
[9] TorrenFreak. Virgin Media CEO Says Net Neutrality is "A Load of Bollocks". http://torrentfreak.com/virgin-media-ceo-says-net-neutrality-is-a-load-of-bollocks-080413/ (April 2008)
[10] The Register. Californian sues Comcast over BitTorrent throttling. http://www.theregister.co.uk/2007/11/15/comcast_sued_over_bittorrent_blockage/ (April 2008)
[11] Bram Cohen. The BitTorrent Protocol Specification http://www.bittorrent.org/beps/bep_0003.html (April 2008)
[12] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips, "A measurement study of the bittorrent peer-to-peer file-sharing system," Tech. Rep. PDS-2004-007, Delft University of Technology, Apr. 2004.
[13] Frank Dabek, Ben Zhao, Peter Druschel, and Ion Stoica. Towards a common API for structured peer-to-peer overlays. In IPTPS '03, Berkeley, CA, February 2003.
[14] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. Technical Report TR-819, MIT, March 2001.
[15] Ben Zhao, John Kubiatowicz, and Anthony Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, April 2001. 55
[16] J.A. Pouwelse and P. Garbacki and J. Wang and A. Bakker and J. Yang and A. Iosup and D.H.J. Epema and M. Reinders and M. van Steen and H.J. Sips (2008). Tribler: A social-based peer-to-peer system. Concurrency and Computation: Practice and Experience 20:127-138. http://www.tribler.org/

Paper B

Chapter 8

Connectivity Properties of Mainline BitTorrent DHT Nodes

R. Jimenez, F. Osmani, B. Knutsson

In *the 9th IEEE International Conference on Peer-to-Peer Computing 2009 (P2P'09)*
Sept. 9–11, 2009, Seattle, Washington, USA

© 2009 IEEE. Reprinted with permission.

Connectivity Properties of Mainline BitTorrent DHT Nodes

Raul Jimenez Flutra Osmani Björn Knutsson

Royal Institute of Technology (KTH)
ICT/TSLAB
Stockholm, Sweden
{rauljc, flutrao, bkn}@kth.se

Abstract

The birth and evolution of Peer-to-Peer (P2P) protocols have, for the most part, been about peer discovery. Napster, one of the first P2P protocols, was basically FTP/HTTP plus a way of finding hosts willing to send you the file. Since then, both the transfer and peer discovery mechanisms have improved, but only recently have we seen a real push to completely decentralized peer discovery to increase scalability and resilience.

Most such efforts are based on Distributed Hash Tables (DHTs), with Kademlia being a popular choice of DHT implementation. While sound in theory, and performing well in simulators and testbeds, the real-world performance often falls short of expectations.

Our hypothesis is that the connectivity artifacts caused by guarded hosts (i.e., hosts behind firewalls and NATs) are the major cause for such poor performance.

In this paper, the first steps towards testing this hypothesis are developed. First, we present a taxonomy of connectivity properties which will become the language used to accurately describe connectivity artifacts. Second, based on experiments "in the wild", we analyze the connectivity properties of over 3 million hosts. Finally, we match those properties to guarded host behavior and identify the potential effects on the DHT.

1 Introduction

The BitTorrent protocol [6] is widely used in Peer-to-Peer (P2P) file sharing applications. Millions of users[1] collaborate in the distribution of digital content every day. As traditional broadcasters transition to Internet distribution, we can expect this number to increase significantly, which raises some concerns about the scalability and resilience of the technology.

Our work is part of the P2P-Next[1] project, which is supported by many partners including the EBU[2] who claims to have more than 650 million viewers weekly. In the face of such load, scalability and resilience become vital components of the underlying technology.

In BitTorrent, content is distributed in terms of objects, consisting of one or more files, and these objects are described by a *torrent*-file. The clients (*peers*) participating in the distribution of one such object form a *swarm*.

A swarm is coordinated by a *tracker*, which keeps track of every peer in the swarm. In order to join a swarm, a peer must contact the tracker, which registers the new peer and returns a list of other peers. The peer then contacts the peers in the swarm and trades pieces of data with them.

The original BitTorrent design used centralized trackers, but to improve scalability and resilience, distributed trackers have been deployed and currently exist in two flavors: *Mainline DHT* and *Azureus DHT*. Both of them are based on Kademlia[11], a distributed hash table (DHT). Kademlia is also the basis of *Kad*[17], used by the competing P2P application eMule[3].

Kademlia's properties and performance have been thoroughly analyzed theoretically as well as in lab settings. Kademlia's simplicity is one of its strengths, making theoretical analysis simpler than that of other DHTs. Furthermore, simulations such as [9] show that Kademlia is robust in the face of *churn*[4].

When we analyze previous measurements on three Kademlia–based DHTs, we find that Kad, eMule's implementation of Kademlia, has demonstrated good performance [17], while the two Kademlia-based BitTorrent DHTs (Mainline DHT and Azureus DHT) show very poor performance [7]. While lookups are performed within 5

[1] The Pirate Bay alone tracks more than 20 million peers at any given time.

[2] European Broadcasting Union

[3] http://www.emule-project.net/ (last accessed April 2009)

[4] Defined in Section 3

seconds 90% of the time in Emule's Kad, the median lookup time is around a minute in both BitTorrent DHTs.

One of the main differences between Kad and the other two implementations is how they manage nodes running behind NAT or firewall devices. Kad attempts to exclude such nodes from the DHT. On the other hand, neither of the BitTorrent's DHT implementations have such mechanisms.

Evidence suggests that some connectivity artifacts on deployed networks were not foreseen by DHT designers. For instance, Freedman et al. [8] show how non-transitivity in PlanetLab degrades DHT's performance (including Kademlia). These connectivity artifacts exist on the Internet as well, as our experiments will show.

Guarded hosts, hosts behind NATs and firewalls [19], are well-known in the peer-to-peer community for causing connectivity issues. Different devices and configurations produce different connectivity artifacts, including non-transitivity.

This evidence leads us to believe that DHT implementations which consider and counteract guarded hosts' effects are expected to perform better than those that do not.

The ultimate test of this hypothesis would be checking whether guarded host's connectivity artifacts significantly affect Kademlia's lookup performance. In order to do this, we need to understand the characteristics of these connectivity artifacts and their potential effects on the DHT routing. Then, we would be able to carry out an experiment looking for these effects on the actual lookups.

In this paper, we focus on the definition and analysis of these connectivity properties. We also underlay the potential effects on the DHT performance. Although we do not attempt to test whether guarded hosts actually degrade lookup performance, an outline of the future work is provided in Section 6.

Mainline DHT, used for BitTorrent peer discovery, was integrated into Tribler[13] —the integral component of the P2P-Next project. The ultimate goal of this ongoing research is to adapt the Mainline DHT to the non-ideal Internet environment, while keeping backward compatibility with the millions of nodes already deployed. Thus, we focus our experiments on the Mainline BitTorrent DHT nodes.

We model nodes' connectivity according to three properties: *reciprocity*, *transitivity*, and *persistence*. This taxonomy in itself is one of the contributions of this paper, since it provides the language needed to reason about and specify the connectivity assumptions made by DHT designers and deployers. For every property, we discuss the possible cause and its potential effects on Kademlia.

In our experiments, the connectivity properties of more than 3 million DHT nodes are studied. We find that most of the connectivity patterns observed correlate to common NAT and firewall configurations.

The following section provides an overview of Kademlia and guarded hosts. In Section 3 the potential effects of the connectivity artifacts are discussed. We present our experiment in Section 4, discuss the results in Section 5, outline the future work in Section 6 and conclude in Section 7.

2 Background

In this section we give the background needed to understand the experiments and the results. We provide basic information regarding Kademlia's routing table management and its lookup routing algorithm. In the second part, we overview the generic behavior found on most common configurations of NATs and firewalls.

2.1 Kademlia

Kademlia[11] is a DHT design which has been widely deployed in BitTorrent and other file sharing applications. When used as a BitTorrent distributed tracker, Kademlia's *objectIDs* are torrent identifiers and *values* are lists of peers in the torrent's swarm.

Each node participating in Kademlia obtains a *nodeID*, whereas each object has an *objectID*. Both identifiers consist of a 160-bit string. The value associated to a given objectID is stored on nodes whose nodeIDs are closest to the objectID, where closeness is determined by performing a XOR bitwise operation on the nodeIDs and objectID strings.

Every node maintains a routing table. The routing table is organized in *k-buckets*, each covering a certain region of the 160-bit key space. Each k-bucket contains up to *k* nodes, which share some common prefix of their identifiers. New nodes are discovered opportunistically and inserted into the appropriate buckets as a side-effect of incoming queries and outgoing lookup messages.

Kademlia makes use of iterative routing to locate the value associated to the objectID, which is stored on the nodes whose nodeIDs are closest to the objectID. The node performing the lookup queries nodes in its routing table — those whose nodeIDs are closest to the objectID. Each of those nodes returns a list of nodes whose nodeIDs are closer to the objectID. The node continues to query newly discovered nodes until the result returned is the value associated to the objectID. This value, when using Kademlia as a BitTorrent tracker, is a list of peers.

2.2 Guarded Hosts

NATs and firewalls are important components of the network infrastructure and are likely to continue to be deployed. According to a recent paper [12], two thirds of all peers are behind NATs or firewalls in open BitTorrent communities. Despite the fact that different firewalls and NATs

can have different configurations, the most common types are overviewed in this subsection.

Note that we just focus on UDP connectivity since the Mainline BitTorrent DHT uses UDP as transport protocol.

2.2.1 Firewalls

A guarded host located behind a firewall is able to send outgoing packets but may be unable to receive incoming packets. Though several firewall configurations are deployed, in this paper, we consider the simplest case where outgoing packets are forwarded but the incoming packets are dropped. In such a scenario, the connectivity is *non-reciprocal*, and the internal host is able to send but not receive any packet.

2.2.2 NAT Behavior

NAT behavior is more complex. A host behind a NAT is able to send packets to hosts on the other side of the NAT. The NAT device, in turn, keeps track of the packets sent by the internal host in its table, in the form of entries that expire within a certain timeout. When the external host replies, the NAT box checks the reply against its address translation table, before routing the reply back to the internal host. For as long as the entry remains in the NAT table, the two hosts are able to communicate, and the communication, according to our property definitions in Section 3, is said to be *persistent*.

The entries in the NAT table are either removed when they timeout or when new entries replace the old ones. Since the DHT nodes contact many other nodes, it is expected that NAT tables can fill up rather quickly.

2.2.3 NAT Timeouts

Recent measurements of NAT/firewall characteristics in the Tribler system[5] reveal that the average NAT timeout value is 2 minutes for more than 60% of the NATed hosts studied. Moreover, the IETF RFC 4787 [3] specifies that a NAT UDP entry should not expire in less than two minutes; it also recommends a default value of 5 minutes or more for each entry. However, since NAT behavior is not really standardized, applications must be extremely conservative, in order to cope with the large variation of (observed) behaviors.

When the entries are removed from the table, the external hosts are unable to reach the internal host, since NAT boxes discard all incoming packets for which they find no match in their table entries. From the perspective of an external host, the internal host is no longer reachable, while in fact, the internal host continues to listen behind the NAT box. Consequently, the size-limited tables or short timeouts of NAT devices may break *persistence*.

[5]https://www.tribler.org/trac/wiki/NATMeasurements (last accessed June 2009)

2.2.4 NAT Configuration

Usually, the NAT (or firewall) device behind which the node is sitting is under the control of the user. Most of the issues created by them can be resolved, or at least mitigated to a large extent, by proper configuration. In many cases, this is as simple as enabling Universal Plug and Play-support[2] (UPnP) in the NAT-box, and have the DHT implement a UPnP-client to correctly setup forwarding.

Alternatively, the DHT application could provide the information needed by the user to manually configure the NAT to forward UDP traffic.

2.2.5 STUN

When participating in the DHT, a node will keep a routing table with pointers to other DHT nodes. Additionally, it will be a tracker for a small number of BitTorrent objects. The role of a node in DHT is to receive queries from other nodes —either updating routing information or performing DHT lookups. In either case, this is a very light weight computational operation.

Session Traversal Utilities for NAT[16] (STUN) may initially seem like an option for dealing with NAT traversal. STUN is certainly possible to implement, and perfectly reasonable for setting up VoIP streams and other long-term communications. However, unlike VoIP streams, DHT messages are very short-term communications (usually a single query/response) and the number of connections to different nodes is high (commonly a few hundred).

For the DHT as a whole, we argue that the cost of using STUN to reach an otherwise unreachable node exceeds the benefit gained by having that node participate in the DHT.

3 Dissecting Churn — Property Definitions

In a DHT, any node can join or leave the DHT at any moment. *Churn* is measured as the number of nodes joining and leaving the DHT during a given period of time, and is thus an indicator of how dynamic a DHT network is.

Since each node needs to keep its routing table updated and accurate, a maintenance overhead is associated with churn. That is why counteracting churn is so important when designing and deploying a DHT.

Much research has been done on DHT performance in presence of churn [15, 18]. Our hypothesis, however, is that a large fraction of the reported churn in deployed DHTs is caused by connectivity artifacts. Unlike real churn, this *apparent churn* follows certain patterns which may be used to identify it and, potentially, eliminate it.

Although we have not attempted to perform similar experiments on other Kademlia-based implementations, we expect that our findings hold for all implementations which

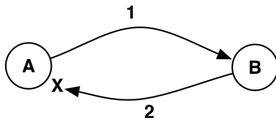

Figure 1. Non-reciprocal connectivity

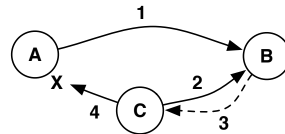

Figure 2. Non-transitive connectivity

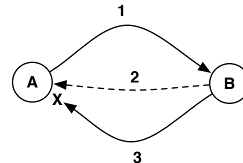

Figure 3. Non-persistent connectivity

do not have explicit mechanisms in place to mitigate guarded nodes' effects on the DHT.

In the next subsections, we define the three connectivity properties we have identified, along with the percentage of DHT nodes exhibiting them in the measurements we have performed. We also examine plausible reasons why a large fraction of nodes are missing one or more of these properties, and how this will impact the performance of the DHT.

Throughout the subsections, the numbers accompanying the protocol descriptions refer to the message exchange order and match the numbers in the corresponding figures.

3.1 Non-reciprocal Connectivity May Create Apparent Churn

On the open Internet, it is assumed that if node A can establish a connection to node B, then node B can establish a connection to node A, i.e., connectivity is *reciprocal*. Our experiments, however, reveal that just **80%** of the nodes exhibit reciprocal connectivity. Firewalls and NATs which forward outgoing, but drop incoming, packets are the likely cause.

Figure 1 depicts the non-reciprocity of the connectivity between A and B. In this scenario, A is the node behind a firewall and the one to initiate the connection with B (1). B assumes the connectivity to be reciprocal and thus inserts A in its routing table.

After a while, when refreshing the buckets in the routing table, node B finds that A no longer replies to its queries. After several failed attempts to reach A (2), B regards A as unreachable and therefore removes it from the routing table.

After being removed from the routing table, A may send a new query to B. As before, B would consider A a good candidate for its routing table and therefore start the process over again.

3.2 Non-transitive Connectivity May Break Lookup Routes

On the Internet, there is a general assumption of *transitivity*, meaning that if node A can reach node B, then any node that can reach B will also be able to reach A. NATs and firewalls break this assumption, and in fact, less than **40%** of the nodes analyzed have transitive connectivity.

Figure 2 illustrates the case, where node A, which is located behind a NAT, causes non-transitive connectivity. When node A sends a query to B (1), B replies back and adds A to its routing table. Later, when node C is performing a lookup, it queries B (2) and B replies with a reference node from its routing table (3), which in this case is A. On the next lookup step, node C sends a query to A (4), but receives no reply. If the connectivity were transitive, C would have been able to reach A, but in this case, C will eventually wait for a timeout —confirming that A is unreachable— and attempt to use an alternate node. Or, formally expressed: C can reach B (2), B can reach A (reply to 1), but C is not able to reach A.

DHTs employing iterative routing, such as Kademlia, are affected by non-transitive connectivity. Concretely, non-transitive connectivity breaks lookup routes.

3.3 Non-persistent Connectivity May Create Apparent Churn

Persistence is a more vague concept. We say that A node exhibits persistent connectivity if it can be reached all the way from the moment it joins until it leaves the DHT.

As explained in Section 2.2, NATs are known to cause ephemeral connectivity. In Figure 3, the connectivity between node A and B is a non-persistent one, where A is located behind a NAT.

Immediately after node A sends a message to B (1), node A is able to receive messages from B (2). Assuming that A does not leave the DHT, B should be able to reach A at any given time. In this case, however, the connection breaks down and node B is unable to reach A (3).

This behavior could be explained in terms of generic NAT behavior, as described in Section 2.2. The NAT enables connectivity between A and B, but only for the period

of time when the entry in the NAT table is valid, after which the connection is effectively broken.

In our experiments, slightly less than **44%** of the nodes were reachable during, at least, a five minute window. Please note that some of the unreachable nodes could be legitimately unreachable, i.e., due to actually leaving. Similarly, some of the reachable nodes may have been reachable only because of communication from other nodes "refreshed" the relevant NAT table entry.

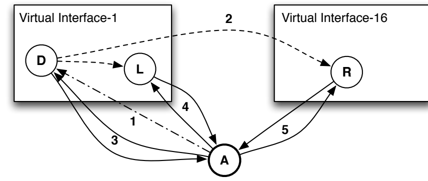

Figure 4. Experiment setup

4 Experiment Description

In this experiment, DHT nodes' reachability is analyzed from three different vantage points. Every time a node sends a query to one of our instrumented DHT nodes, queries are sent from (1) the same IP and port number, (2) same IP but different port number, and (3) a different IP.[6] The process is repeated after a period of 5 minutes.

The pieces of software developed are described in the following subsections. Both source code and result logs are available online.[7]

The setup consists of one PC running Ubuntu GNU/Linux. This computer is assigned 17 IP addresses which are managed through virtual interfaces. *remotechecker* is associated with one of the virtual interfaces. While an *instrumented DHT node* and a *localchecker* are associated with each of the rest of virtual interfaces.

Our DHT nodes' identifiers are chosen in a way that the first four bits are different from each other. This "spreads out" our nodes in the identifier space. The aim of this configuration is to broaden the DHT identifier space coverage in order to discover as many nodes as possible.

Figure 4 illustrates the reachability analysis process. Numbers in the arrows indicate chronological order and are referenced throughout the following subsections.

4.1 Rechability Checker

We have developed a piece of software called Reachability Checker (*RChecker*). Rchecker checks and logs reachability information regarding a given DHT node.

Nodes are identified by their IP address and nodeID. Nodes with different nodeIDs and same IP could be different nodes running on the same host or on different hosts behind a common NAT. Nodes with the same nodeID and different IP should not exist on the DHT. The latter nodes exist, albeit in small numbers, and are considered in the results. When two queries are received from the same IP and nodeID but different port, they are considered as coming from the same node, and just the first instance is considered.

[6] The reference point is our modified DHT node (IP address and port).
[7] http://tslab.ssvl.kth.se/raul/p2p09/

Subsequent queries from nodes that were already checked are discarded.

Every time a node is to be checked, RChecker sends a burst of queries to the node. Queries are sent every 5 seconds, up to 5 times. Once RChecker receives a reply, a *reachable* status is recorded and no more queries are sent. This multiple querying should avoid recording reachable nodes as unreachable due to temporary network conditions.

If no reply is received within one minute, an *unreachable* status is recorded. Note that most of the BitTorrent Kademlia implementations have a 20 seconds timeout. Some have argued that 20 seconds is already too long and actually harms lookup performance [7]. In this experiment, however, we have chosen such a long timeout because we want to be able to detect network connectivity; even when the round trip time is longer than a DHT implementation's timeout would be.

A second burst, identical to the one described above, is sent 5 minutes later.

4.2 Instrumented DHT Node

We have instrumented Tribler's implementation of Kademlia[8]. The original code is modified to call RChecker as needed. Additionally, the socket used by Kademlia is passed to RChecker, so the queries are sent using the same source IP and port.

Everytime a query is received (1) and the node has not been already checked, the node's information (IP, port, ID) is sent to localchecker and remotechecker (2). Then, RChecker is called in order to check the node's reachability using the same IP and port as the DHT node (3).

4.3 Localchecker

Every localchecker is listening to the instrumented DHT node sharing the same virtual interface —i.e., both have the same IP address. Every time localchecker receives information from the instrumented DHT node (2), it calls RChecker to check the node's reachability from the same IP address as the DHT node but different port (4).

[8] http://svn.tribler.org/khashmir/ (last accessed June 2009)

Table 1. Experiment results and possible causes

Pattern	Nodes (%)	Possible cause(s)
UUU−UUU	10.6	Firewall
RUU−UUU	**31.3**	Port restricted cone
RUU−RUU	2.8	and symmetric NAT
RRU−UUU	0.8	Restricted cone NAT
RRU−RRU	2.0	
RRR−UUU	2.7	Full cone NAT and real churn
RRR−RRR	**35.5**	Open Internet
UUU−RRR	7.6	Behavior not matched
RRU−RRR	1.7	
Other	5	Rest of the cases

4.4 Remotechecker

The single remotechecker listens to every instrumented DHT node. Every time remotechecker receives information from the DHT node (2), it calls RChecker to check the node's reachability from an IP address which is different from the one used by the instrumented DHT node (5).

5 Experiment Results

During 24 hours of running the experiment, 3,683,524 unique nodes were observed. Table 1 shows the observed connectivity patterns along with the NAT or firewall types, matching the pattern and the percentage of nodes.

The notation used throughout this section is `XXX−XXX`, where the `X` can be either `R` (reachable) or `U` (unreachable). The connectivity fingerprint of each checked node can be represented by this 6-character string.

The first character accounts for the reachability of the node from the instrumented DHT node (same IP and same port). The second character represents the reachability of the node from *localchecker* (same IP but different port). Likewise, the third one indicates wheather the node is reachable or not from the *remotechecker* (different IP).

The last three characters follow the same structure, however, they represent node's connectivity after a 5 minute period.

5.1 Analysis

More than **10%** of the nodes are globally unreachable (`UUU−UUU`). They are able to send messages — our modified DHT node received at least one query from them. They are, however, unable to receive messages from any of our vantage points. This connectivity pattern matches the firewall behavior, configured to let outgoing messages through but drop incoming messages.

A large percentage of the nodes in the DHT population are only partially reachable. Typically, they can be reached only under certain circumstances. We argue that NATs are the main cause of this partial reachability of nodes.

As explained in Section 2.2, NAT devices have a timeout parameter which make stale entries expire after a given period of time. In table 1, NAT types have two associated observed patterns. The former matches the case when the NAT entry expires within 5 minutes, therefore, the node is unreachable the second time it is checked. In the latter, the NAT timeout is longer than 5 minutes. Notice that a *full cone* NAT, whose entry has not expired, matches the open internet behavior.

More than **34%** of the nodes are reachable from our modified DHT node, but neither from *localchecker* nor *remotechecker* (`RUU−RUU` and `RUU−UUU`). This behavior matches *port restricted cone* and *symmetric* NAT types. These NAT types register outgoing connections that are initiated by an internal host. An incoming packet is only forwarded to the internal host if both the IP and port of the external host match the NAT's entry. Packets coming from the same IP but different port (*localchecker*) or a different IP (*remotechecker*) are discarded by the NAT device.

About **3%** of the nodes are reachable from our modified DHT node and the *localchecker* but not from the *remotechecker* (`RRU−UUU` and `RRU−RRU`). The plausible explanation is that the node is behind a *restricted cone* NAT, in which case, the incoming packets are forwarded only when the IP address of the external host matches the NAT's entry. Therefore, packets coming from our modified DHT node and the *localchecker* (same IP) are received by the analyzed node, while those from the *remotechecker* are dropped.

Less than **3%** of the nodes in the DHT are reachable from the instrumented DHT, the *localchecker* and the *remotechecker* during the first time when testing their connectivity (`RRR−UUU`). However, the nodes are globally unreachable when checked after a period of 5 minutes. The probable cause of this pattern is a full cone NAT, whose corresponding entries in the NAT table have expired within the testing period. This case will be further discussed in this section.

Approximately **35.5%** of the DHT nodes are globally reachable. They are reachable from all of our vantage points before, as well as, after the 5 minute waiting period.

Finally, we show two patterns that do not match any of the expected behaviors but represent more than 1% each. They are `UUU−RRR` (7.6%) and `RRU−RRR` (1.7%). These cases remain open for further research.

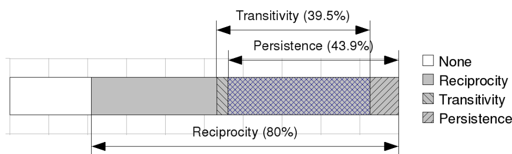

Figure 5. Properties Chart

5.2 Correlation between Transitivity and Persistence

Figure 5 depicts transitivity (RXR–XXX) and persistence (RXX–XXX) as subsets of reciprocity (RXX–XXX). We also notice the significant overlap between transitivity and persistence. This overlap was indeed expected since NAT devices commonly cause both non-transitivity and non-persistence, as previously discussed. Furthermore, some of the cases exhibiting persistence and non-transitivity might be due to long NAT table timeouts or casual messages – "refreshing" the right NAT table entry.

Based on the above observations, we can formulate the heuristic that if a node's connectivity is known to be transitive, it is very likely to be persistent as well, and vice versa. By applying this heuristic, we may be able to use a less expensive method of testing connectivity, without a significant loss of accuracy.

5.3 Apparent & Real Churn

Since we identify the nodes' properties from an outsider's point of view, we do not know what connectivity properties a given node actually has. This fact complicates the task of differentiating apparent churn from nodes effectively leaving the DHT (i.e., real churn).

We can certainly say that any node which replies to one or more of our venture points after the 5 minute period, has not left the DHT. Therefore, connectivity patterns in the XXX–UUU category might be caused by nodes actually leaving the DHT. This category accounts for 45.6% of the nodes.

The UUU–UUU pattern (10.6%) belongs to this category. These nodes fail to reply to us immediately after they have sent us a query[9]. Since the time is so short (a UDP round trip) we can assume that very few nodes, if any, would have left the DHT within such extremely short period. Instead, we argue that this is apparent churn caused by firewalls.

Another interesting pattern is RRR–UUU (2.7%). This pattern may be caused by *full cone* NAT which forwards the traffic to the internal host regardless of the source's IP address, but the NAT entry would expire within the 5-minutes window. However, according to our observations, DHT nodes constantly receive and send messages which refresh the NAT entries, thus making the connections effectively open, given a long enough NAT timeout. This fact makes us believe that a good part of these cases corresponds to real churn —i.e., nodes in the open Internet which have left the DHT within the 5-minutes window.

The case which accounts for most of the nodes in the XXX–UUU category is RUU–UUU (31.3%). The fact that these nodes have limited connectivity in the first place makes them unfit to carry out DHT tasks. Therefore, DHT implementations that avoid adding nodes with limited connectivity into the routing tables, will most likely not experience the churn issues caused by NATs, notoriously reducing DHT's churn.

[9]The query triggers the reachability check.

6 Related and Future Work

6.1 Dealing with Limited Connectivity Nodes

As previously stated, our hypothesis is that limited connectivity often is a result of an improperly configured NAT/firewall. The very first step towards dealing with limited connectivity should thus be to properly document the requirements of the client, and to make it easy to configure and test port forwarding in the NAT/firewall for the client.

Still, the DHT must be able to cope with the problems limited connectivity nodes pose, and we have seen in Emule's Kad [17] that fairly simple modifications to existing DHT implementations can go a long way towards mitigating the effects of limited connectivity.

Many of the proposals and perfomed simulations have mainly tried to mitigate the negative effects and improve the overall performance of the DHT, but none has addressed the underlying problem. Moreover, their benefits come at the cost of other performance factors, mainly bandwidth consumption. We find examples of such improvements in [7, 4, 9]:

- Check node's reachability before adding it to the routing table.
- Reduce timeout value or implement adaptive timeouts.
- Increase lookup parallelism.
- Increase the refresh rate, such that dead nodes are discovered earlier.
- Implement an "extended table" or bigger size routing tables, such that the probability of having fresh entries in the routing table increases.

- Maintain small and fresh routing tables, by removing neighbors whose estimated probability of being alive is below some calculated threshold [10].

The above parameter fine-tunings should be considered in the widely-deployed Kademlia DHT, where millions of users simultaneously participate today and tens of millions of users may participate in the near future. The increase in user participation implies larger routing tables, and a potentially exponential growth in the number of maintenance messages. The proposed tweaks requiring additional messages would exacerbate this growth, and may prove an obstacle to DHT scalability. Tweaks that only require local resources, i.e., memory and processing, are much more likely to scale, and will benefit from Moore's Law.

Another approach is to try to determine the specific properties of hosts before adding them to the routing table, see the discussion in Section 3. Rhea proposes a set of measurements in order to counteract the effects of non-transitive connectivity on OpenDHT [14].

As our experiment demonstrates, the connectivity properties essential to a DHT of a given node can be determined. Reciprocity is easily detected by sending a single query, while checking transitivity is more complex to detect. The strategy used in our experiments relied on multiple IP addresses being available to the test host, but we can't expect a normal DHT node to have access to more than a single IP address.

While one DHT node could use another node as *remotechecker*, letting it relay queries and report the reachability status, this opens a whole new can of worms. For example, this mechanism could be exploited for DDoS[10] attacks.

Nevertheless, *localchecker* can be easily implemented and deployed without the need of several IP addresses per host or additional trust. In fact, *localchecker* is able to correctly identify most of the non-transitive connectivity cases. Based on our results, only 4.6% of them are detected by *remotechecker* but not by *localchecker*. Thus, if only the local mechanisms were to be applied, we would still vastly improve the quality of nodes in the routing table. Furthermore, we would avoid introducing excessive complexity and security vulnerabilities.

An additional mechanism that would improve the quality of the routing table content is to quarantine new nodes before adding them to the routing table. This gives enough time to perform a second reachability check in order to determine whether the candidate node's connectivity is persistent, similarly to the approach used in our experiment.

As seen previously in Figure 5, transitivity and persistence are correlated but do not completely overlap. Therefore, either *localchecker* or quarantine alone would identify most of the limited connectivity nodes, but a combination of both would correctly classify the vast majority of nodes, thus increasing the detection effectiveness.

[10]DDoS stands for Distributed denial of service.

The mechanisms described above can be combined with a policy that is consistent with our discussion in 2.2.5 — where guarded nodes are not allowed to join the DHT. In Kad, however, the node will instead find a DHT node to use as a proxy. While this approach has been used in a fairly large deployment, it moves load and responsibility to nodes already in the DHT, thus adding complexity by requiring a separate proxy mechanism/protocol.

A policy similar to the one used in StealthDHT[5] might be more appropriate. According to StealthDHT, nodes participating in the DHT are separated in two categories: *service nodes* and *stealth nodes*. Service nodes perform routing and value storage tasks, while stealth nodes are not involved in any active task but are able to maintain their own routing tables and perform lookups.

We would like to take a further step and add conceptual as well as practical separation. Nodes which are able to, will be part of the DHT and act as a *service node*, handling routing and storage of values. Nodes with limited connectivity will only be clients. As such, they will perform their own lookups using DHT nodes, and they may even cache information locally, but they will never be contacted by other nodes. Finally, notice that, due to Kademlia's iterative routing, service nodes only need to reply to simple queries, while DHT clients can initiate and keep track of the lookup's state on their own.

6.2 Future Work

We have argued that the large percentage of DHT nodes having limited connectivity has repercussions on the DHT performance. They become passive participants of the routing tables, only causing delays and stale entries.

At the most basic level, Emule's Kad implementation tries to detect nodes that don't reply to queries, and completely excludes them from the DHT. However, since we can find no references to how or why this was done, we are unable to determine whether this was an *ad hoc* solution, or it was the result of careful design based on a study similar to ours.

In this paper, we have studied the connectivity properties of nodes by deploying a set of DHT nodes and studying the properties of nodes which exchange messages with them. However, the fact that guarded hosts exist, and are active in the DHT, is not a problem per se. It is only when routing tables are effectively poisoned that lookup performance declines. A logical next step would thus be to take inventory of the routing tables of DHT nodes "in the wild", and find out to what extent guarded nodes actually end up in routing tables.

Furthermore, our ultimate goal is to use the knowledge we have gained from this research to repair and improve the DHT performance. As dissussed in the previous subsection, that would include designing mechanisms that identify/remove limited connectivity nodes from the routing table, and prevent such nodes from being added in the first place.

Finally, it could be instructive to compare the "pollution rate"[11] in the routing tables of different DHTs, such as the Mainline and Azureus DHT, as well as eMule's Kad.

7 Conclusion

In this paper, we have defined a set of properties which provides the language needed to spell out the assumptions made by DHT designers and deployers. These properties were not explicitly considered in the original Kademlia design. Instead, their effects were only discovered when DHTs were deployed and faced with the non-ideal connectivity artifacts in the real world.

We have studied over 3 million BitTorrent Mainline DHT nodes' connectivity according to these properties. The results point to the generalized presence of NAT and firewall devices causing connectivity issues in the DHT. In fact, only around one third of the nodes analyzed have "good connectivity" —i.e. reciprocal, transitive, and persistent.

Finally, we do not propose a stopgap solution for poor DHT performance. Instead, we offer the taxonomy to explicitly specify the DHT's connectivity assumptions and the toolkit to determine whether those assumptions are met. Our long-term ambition is to enable ourselves and others to design and implement DHTs where the underlying problems are addressed, instead of just tweaking parameters and adding kludges to handle the symptoms.

Acknowledgment

The research leading to these results has received funding from the Seventh Framework Programme (FP7/2007-2013) under grant agreement No. 21617 (P2P-Next).

We would like to thank Lucia d'Acunto, Delft University, The Netherlands, for providing us with preliminary results of her experiments on NATs.

References

[1] P2P-Next Project. http://www.p2p-next.org/ (last accessed June 2009).

[2] UPnP Forum. Internet Gateway Device (IGD) V 1.0. http://www.upnp.org/ (last accessed June 2009).

[3] F. Audet and C. Jennings. Network Address Translation (NAT) Behavioral Requirements for Unicast UDP. Technical report, BCP 127, RFC 4787, January 2007.

[4] A. Binzenhfer, H. Schnabel, A. Binzenhfer, and H. Schnabel. Improving the performance and robustness of kademlia-based overlay networks, 2007.

[5] A. Brampton, A. MacQuire, I. A. Rai, N. J. P. Race, and L. Mathy. Stealth distributed hash table: a robust and flexible super-peered dht. In *CoNEXT '06: Proceedings of the 2006 ACM CoNEXT conference*, pages 1–12, New York, NY, USA, 2006. ACM.

[6] B. Cohen. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, volume 6. Berkeley, CA, USA, 2003.

[7] S. A. Crosby and D. S. Wallach. An analysis of bittorrent's two kademlia-based dhts, 2007.

[8] M. J. Freedman, K. Lakshminarayanan, S. Rhea, and I. Stoica. Non-transitive connectivity and dhts. In *In Proc. of the 2nd Workshop on Real Large Distributed Systems*, 2005.

[9] J. Li, J. Stribling, T. M. Gil, R. Morris, and M. F. Kaashoek. Comparing the performance of distributed hash tables under churn. In *In Proc. IPTPS*, 2004.

[10] J. Li, J. Stribling, R. Morris, and M. F. Kaashoek. Bandwidth-efficient management of dht routing tables. In *NSDI*, 2005.

[11] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. pages 53–65, 2002.

[12] J. Mol, J. Pouwelse, D. Epema, and H. Sips. Free-Riding, Fairness, and Firewalls in P2P File-Sharing. In *Proceedings of the 2008 Eighth International Conference on Peer-to-Peer Computing-Volume 00*, pages 301–310. IEEE Computer Society Washington, DC, USA, 2008.

[13] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. J. Epema, M. Reinders, M. R. van Steen, and H. J. Sips. Tribler: a social-based peer-to-peer system: Research articles. *Concurr. Comput. : Pract. Exper.*, 20(2):127–138, 2008.

[14] S. Rhea. *OpenDHT: A Public DHT Service*. PhD thesis, UNIVERSITY OF CALIFORNIA, 2005.

[15] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a dht. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[16] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. STUN-simple traversal of user datagram protocol (UDP) through network address translators (NATs). Technical report, March 2003. RFC 3489.

[17] M. Steiner, T. En-Najjary, and E. W. Biersack. A global view of kad. In *IMC '07: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 117–122, New York, NY, USA, 2007. ACM.

[18] D. Stutzbach and R. Rejaie. Understanding churn in peer-to-peer networks. In *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 189–202, New York, NY, USA, 2006. ACM.

[19] W. Wang, H. Chang, A. Zeitoun, and S. Jamin. Characterizing guarded hosts in peer-to-peer file sharing systems. In *IEEE GLOBECOM Global Internet and Next Generation Networks Symposim*, 2004.

[11] Number of limited connectivity nodes versus the total number of nodes in the routing table.

Paper C

Chapter 9

Sub-Second Lookups on a Large-Scale Kademlia-Based Overlay

R. Jimenez, F. Osmani, B. Knutsson

In *the 11th IEEE International Conference on Peer-to-Peer Computing 2011 (P2P'11)*
Aug. 30 – Sept. 2, 2011, Kyoto, Japan

© 2011 IEEE. Reprinted with permission.

Sub-Second Lookups on a Large-Scale Kademlia-Based Overlay

Raul Jimenez, Flutra Osmani and Björn Knutsson
KTH Royal Institute of Technology
School of Information and Communication Technology
Telecommunication Systems Laboratory (TSLab)
{rauljc, flutrao, bkn}@kth.se

***Abstract*—Previous studies of large-scale (multimillion node) Kademlia-based DHTs have shown poor performance, measured in seconds; in contrast to the far more optimistic results from theoretical analysis, simulations and testbeds.**

In this paper, we unexpectedly find that in the Mainline BitTorrent DHT (MDHT), probably the largest DHT overlay on the Internet, many lookups already yield results in less than a second, albeit not consistently. With our backwards-compatible modifications, we show that not only can we reduce median latencies to between 100 and 200 ms, but also consistently achieve sub-second lookups.

These results suggest that it is possible to deploy latency-sensitive applications on top of large-scale DHT overlays on the Internet, contrary to what some might have concluded based on previous results reported in the literature.

I. Introduction

Over the years, distributed hash tables (DHTs) have been extensively studied, but it is only in the last few years that multimillion node DHT overlays have been deployed on the Internet. To our knowledge, only three DHT overlays (all of them based on Kademlia [1]) consist of more than one million nodes: Mainline DHT (MDHT), Azureus DHT (ADHT), and KAD. The first two are independently used as trackers (peer discovery mechanisms) by BitTorrent [2], while KAD is used both for content search and peer discovery in eMule (a widely used file-sharing application).

KAD has been thoroughly studied [3], [4], [5]. Stutzbach and Rejaie [3] reduced median lookup latency to approximately 2 seconds (with a few lookups taking up to 70 seconds). Steiner et al. [4] focused solely on lookup parameters, providing useful correlations between parameter values and lookup latency. Their modest achieved lookup performance (lowest median lookup latency at 1.5 seconds), authors discovered, was not due to shortcomings in Kademlia or the KAD protocol but due to limitations in eMule's software architecture.

In this paper, we focus on Mainline DHT which, with up to 9.5 million nodes[1], is probably the largest DHT overlay ever deployed on the Internet [6]. In 2007, a study of the then most popular node implementation in MDHT reported median lookup latencies around one minute [7] and, to our knowledge, no systematic attempts to improve lookup performance have been reported.

Lookup latency results for MDHT, KAD and ADHT [7], [8] are disappointing considering the rather promising latency figures —in the order of milliseconds— previously reported in studies using simulators and testbeds [9], [10]. Our main goal is to improve lookup performance in MDHT, thus closing the gap between simulators and real-world deployments.

To measure and compare performance, we developed a profiling toolkit able to measure different node properties (lookup performance and cost among others) by parsing the network traffic generated during our experiments. This toolkit is capable of profiling any MDHT node, including closed-source, without the need of its instrumentation.

We profiled the closed-source μTorrent (also called UTorrent) implementation, currently the most prevalent MDHT implementation with 60% of the nodes in the overlay. Our results show that UTorrent's performance is unexpectedly good, with median lookup latencies well under one second.

UTorrent's performance does not, however, fulfill the demands of latency-sensitive applications such as the system that motivated this work (see Section II) because more than a quarter of its lookups take over a second. Thus, in an attempt to further reduce lookup latency, we developed our own MDHT node implementations.

In this paper, we show that our best node implementations achieve median lookup latencies below 200 ms and sub-second latencies in almost every single lookup, meeting our system's latency requirements.

The rest of the paper is organized as follows. The background is presented in Section II. Section III introduces the profiling toolkit. Section IV describes our MDHT node implementations while Section V discusses routing modifications. Section VI presents the experimental setup, Sections VII and VIII report the results obtained, Section IX briefly presents related work, and Section X concludes.

II. Background

The work presented in this paper is part of the P2P-Next project[2]. This project's main aim is to build a fully-distributed content distribution system capable of streaming live and on-demand video. Unlike file sharing applications, this is an interactive application, and thus reducing perceived latency

[1] A real-time estimation is available at http://dsn.tm.uni-karlsruhe.de/english/2936.php (June 2011)

[2] http://p2p-next.org/ (June 2011)

(e.g., the time it takes to start playback of a video after the user selects it) to a level acceptable by users is of great importance [11].

This system uses BitTorrent [2] as transport protocol and a DHT-based mechanism to find *BitTorrent peers* in a *swarm*. To avoid confusion, the following terms are defined here: *BitTorrent peers* (or simply *peers*) are entities exchanging data using the BitTorrent protocol; a *swarm* is a set of peers participating in the distribution of a given piece of content; and *DHT nodes* (or *nodes* for short) are entities participating in the DHT overlay and whose main task is to keep a list of peers for each swarm.

Our work is not, however, restricted to BitTorrent or video delivery. One can imagine more demanding systems, for instance, a DHT-based web service capable of returning services (e.g. a web page) quickly and frequently. CoralCDN [12] is a good example of such a service, although its scale is much smaller.

Our hope is that our results will encourage researchers and developers to deploy new large-scale DHT-based applications on the Internet.

A. Kademlia

Kademlia [1] belongs to the class of prefix-matching DHTs, which also includes other DHTs like Tapestry [13] and Pastry [14].

In Kademlia, each node and object are assigned a unique identifier from the 160-bit key space, respectively known as *nodeID* and *objectID*. Pairs of (*objectID*, *value*) are stored on nodes whose *nodeID* are closest to the *objectID*, where closeness is determined by performing an XOR bit-wise operation. In BitTorrent, an *objectID* is a swarm identifier (called *infohash*) and a *value* is a list of peers participating in a swarm.

A lookup traverses a number of nodes in the DHT overlay, each hop progressing closer to the target *objectID*. Each node maintains a tree-based routing table, containing $O(log\ n)$ *contacts* (references to nodes in the overlay), such that the total number of lookup hops does not exceed $O(log\ n)$, where n is the network size. The routing table is organized in *buckets*, where each bucket contains up to *k* contacts sharing some common prefix with the routing table's owner. Each contact in the bucket is represented by the triple *(nodeID, IP address, port)*.

New nodes are discovered opportunistically and inserted into appropriate buckets as a side effect of incoming queries and outgoing messages. To prevent stale entries in the routing table, Kademlia replaces stale contacts —nodes that have been idle for longer than a predefined period of time and fail to reply to active pings— with newly discovered nodes.

To locate nodes close to a given *objectID*, the node performing the lookup uses iterative lookup from start to finish. This node queries nodes from its routing table whose identifiers have shorter XOR distances to the *objectID*, and waits for responses. The newly discovered nodes —included in the responses— are then queried during the next lookup step. Kademlia makes use of parallel routing to send several parallel lookup requests, in order to decrease latency and the impact of timeouts. Lookup terminates when the closest nodes to the target are located.

B. Improving Lookup Performance

Given Kademlia's iterative lookup, lookup performance can be greatly enhanced by modifying the initiating node alone, without the need of changing any other node in the overlay. Thus, modified nodes can be deployed at any moment, setting the path for experimentation and incremental deployment of "better", yet backward-compatible, node implementations.

Researchers have proposed various approaches to increase overall lookup performance in iterative DHTs, while keeping costs relatively low. Parallel lookups and multiple replicas are two parameters that have often been fine-tuned to reduce the probability of lookup failures and alleviate the problem of stale contacts in routing tables, which in turn, increase DHT performance. Various bucket sizes, various-length prefix matching (known as symbol size) and reduced —usually RTT-based— timeout values have also been investigated as means of improving the overall performance.

We discuss some of these improvements in detail in Section IV and V, where we present the modifications we have deployed and measured.

III. Profiling MDHT Nodes

In a DHT overlay, nodes are independent entities that collaborate with each other in order to build a distributed service. A DHT protocol defines the interaction between nodes, but provides significant latitude in how to implement it. Indeed, the Mainline DHT protocol specification [15] leaves many blanks for the implementer to fill in as best as he can.

It follows naturally that many different node implementations will coexist in the MDHT overlay. Some, developed by commercial entities (e.g., Mainline and UTorrent), others cooperatively as open source projects (e.g., Transmission and KTorrent). Even though they have been developed to coexist, significant differences in their behavior can be observed, parts just accidents of separate development, others the result of making different trade-offs.

From our initial studies of Mainline DHT, we had observed diversity in the existing MDHT node implementations. We also recognized that our efforts to improve the performance of MDHT nodes would likely make use of the latitude afforded by the protocol specifications, and thus it was of critical importance that we be able to study the impact of our modifications. To this end, we built a toolkit for profiling and analyzing the behavior and performance of MDHT nodes.

A. Profiling Tools

Instrumenting an open source DHT node is a common approach to measure its performance. The instrumented node would join an overlay, perform lookups, and log performance measurements.

It is, however, unpractical to instrument nodes whose source code is unavailable. In MDHT, UTorrent is by far the most

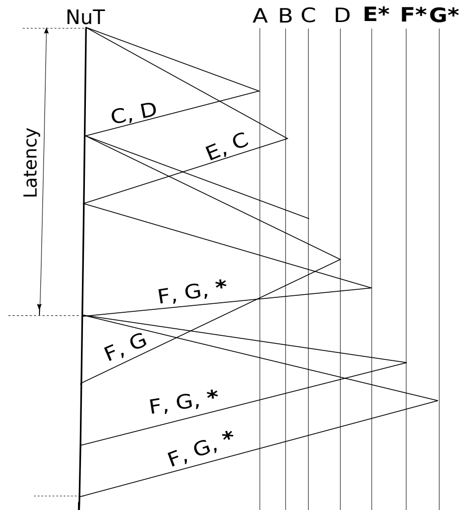

Fig. 1. Lookup performed by node under test (NuT). Letters A–G represent nodes in the overlay. Values are represented with "*".

popular node implementation (2.7 out of 4.4 million nodes according to our results presented in Section VIII). Given that UTorrent's source code is closed, we devised a different approach.

Our toolkit uses a black-box approach: an MDHT node is commanded to perform lookup operations (by using the node's GUI or API), while simultaneously capturing its network traffic. Figure 1 illustrates a lookup performed by the *node under test* (NuT). This node joins the MDHT overlay and is under our control (we can command it to perform lookups); the rest of the nodes shown in the figure (A–G) are a minute fraction of the millions of MDHT nodes —over which we have no control.

Whenever NuT sends or receives a message, the data packet is captured. When the experiment is over, all captured packets can be parsed to measure lookup latency and cost, among other properties.

The toolkit's core, written in Python, provides modules to read network captures and decode MDHT messages. Tools to analyze and manipulate messages are also available, as are the plotting modules that can produce various graphs. Along with the rest of the software presented in this paper, we have released the profiling toolkit under the GNU LGPL 2.1 license.

We use the toolkit in Sections VII and VIII to illustrate our results, and as will be seen, it is already capable of measuring and displaying many interesting properties of MDHT nodes. New measurements and presentations are easily added as plug-ins, using existing analysis and presentation plug-ins as templates.

B. *Profiling Metrics*

In theoretical analysis and simulations of DHTs, lookup performance is often measured in routing hops between the initiator —node performing the lookup— and the node closest to the target key. Although our profiling toolkit can measure hops, we find it more appropriate to measure lookup latency because that is the parameter determining whether a DHT is suitable for latency-sensitive applications.

In this paper, we define **lookup latency** as the time elapsed between the first lookup query is sent and the first response containing values is received.

Figure 1 illustrates a full lookup performed by the node under test. NuT starts the lookup by selecting the nodes closest to the target in its routing table (A and B) and sending lookup queries to them. NuT receives a response from A containing nodes (C and D) closer to the target but no values. Then, NuT sends queries to nodes C and D. The lookup continues until NuT receives a response containing values (*) from E. At this point we consider the goal achieved and we record the lookup latency, although the lookup can progress further to obtain more values associated with the key, as we will discuss in Section VIII.

We define **lookup cost** as the number of lookup queries sent before receiving a value (including queries sent but not yet replied). In the example above, lookup cost is five queries. At the time values are retrieved (E's response contains values), three queries were replied (A, B and E), D replies shortly after, and C never replies (this query would eventually trigger a timeout).

Finally, we define **maintenance cost** as the total number of maintenance queries —`ping` and `find_node` messages— sent by the node under test. These queries are sent to detect stale entries in the routing table and find replacements for these entries. As we later propose modifications to the routing table management, which is the source of maintenance traffic, we will also measure their impact on maintenance cost.

IV. Implementing MDHT Nodes

We have developed a flexible framework based on a plug-in architecture, capable of creating different MDHT nodes. The central part of the architecture handles the interaction between network, API, and plug-ins; while the plug-ins contain the actual policy implementation. There are two categories of plug-in modules: *routing modules* and *lookup modules*. The policies are concentrated on these plug-ins (e.g., all algorithms and parameters related to routing table management are exclusively located in routing modules), simplifying their modification. This architecture allows us to quickly implement different routing table and lookup configurations, and compare them against each other.

The combination of the core, a routing module and a lookup module forms a fully-functional MDHT node, which can be deployed and further analyzed with the profiling tools described in Section III-A.

Although this paper examines only two lookup and four routing modules, several additional modules have been de-

signed, implemented and measured. The modules presented here have been chosen due to their characteristics and their effects on lookup performance and cost.

Even though our plug-in architecture allows us to freely modify lookup modules, we observe that merely adjusting well-know lookup parameters can dramatically improve lookup performance.

These lookup parameters are known as α and β. The α parameter determines how many lookup queries are sent in parallel at the beginning of the lookup, while β is the number of maximum queries sent when a response is received. Figure 1 is an example of a lookup with both parameters set to two.

In this paper, we describe and measure two lookup modules:

- **Standard Lookup** Since the protocol specification does not specify parameters such as α and timeout values, we have resorted to an analysis of UTorrent's lookup behavior. According to our observations, UTorrent's value for α and β are four and one, respectively. Our *standard lookup* implements the same parameters.
- **Aggressive Lookup** In our *aggressive lookup* module, β is set to three while α remains four.

Our routing modules introduce much deeper modifications to the original MDHT routing table management specified in BEP5 (BitTorrent Enhancement Proposal 5) [15]. These modifications are detailed in the next section.

V. ROUTING MODULES

Although some of the previous studies on Kademlia performance have considered modifications on routing table management, most of them estimate the performance gain assuming that all nodes implement them.

We do not propose global modifications where all nodes in the overlay must be modified to obtain benefits. Instead, we propose modifications that benefit the nodes implementing them, regardless of whether other nodes in the overlay implement these modifications or not.

To our knowledge, this is the first attempt to deploy alternative routing table management implementations on an existing multimillion overlay on the Internet, and then measure their effects on lookup latency.

A. Standard Routing Table Management (BEP5)

The *BEP5* routing module aims to implement the routing table management specified in the BEP5 specifications [15] as rigorously as possible. The specifications define bucket size k to be 8. The routing table management mechanism is summarized next.

When a message is received, query or response, the appropriate bucket is updated. If there is already an entry corresponding to this node in the bucket, the entry is updated. Otherwise, three scenarios are possible: (1) if the bucket is full of *good* nodes, the new node is simply discarded; (2) if there is a *bad* node inside the bucket, the new node simply replaces it; (3) if there are *questionable* nodes inside the bucket, they are pinged; if any of them fail to respond after two ping attempts, they will be replaced.

According to the specifications, nodes are defined as *good* nodes if they respond to queries or they have been seen alive in the last 15 minutes. Nodes which have not been seen alive in the last 15 minutes become *questionable*. Nodes that failed to respond to multiple consecutive queries (we chose this value to be two) are defined as *bad* nodes.

Buckets are usually kept fresh as a side effect of lookup traffic. Buckets which have not been opportunistically refreshed in the last 15 minutes are refreshed by performing a maintenance lookup. Maintenance lookups are similar to normal lookups but they use `find_node` messages instead of `get_peers`.

B. Nice Routing Table Management (NICE)

The *NICE* routing module attempts to improve the quality of the routing table by continuously refreshing nodes in the routing table and checking their connectivity. While, as our results show, this quality improvement directly reduces our nodes' lookup latency, we expect other nodes to be also benefited as a side effect. We plan to measure this indirect benefit in future work.

The refresh task is regularly triggered (every 6 seconds in NICE). Each time it is triggered, it selects a bucket and pings the most stale node in the bucket. This continuous refresh guarantees that each bucket must have at least one contact that was recently refreshed and no contacts that have not been refreshed for more than 15 minutes. As a side benefit, this makes maintenance traffic smooth and predictable, with a maximum maintenance traffic of 10 queries per minute.

This module also actively probes nodes to detect and remove nodes with connectivity issues from the routing table. In particular, we implement the quarantine mechanism we previously proposed [16] where nodes are only added to the routing table after a 3 minute period. This quarantine period is mainly aimed at detecting DHT nodes with limited connectivity (probably caused by nodes behind NAT and firewall devices) which cause widespread connectivity artifacts in Mainline DHT, hindering performance.

C. NICE + Low-RTT Bias (NRTT)

In Kademlia, any node falling within the region covered by a bucket is eligible to be added to that bucket. Kademlia follows a simple but powerful strategy of preferring nodes that are already in the bucket over newly discovered candidates. The reasoning is that this policy leads to more stable routing tables [1].

Having stable contacts in the routing table benefits lookups by reducing the probability of sending lookup queries to nodes that are no longer available. Likewise, if the round trip time (RTT) to these nodes is low, then the corresponding lookup queries will be quickly responded, reducing lookup latency.

The impact of low-RTT bias in routing tables has been previously discussed [17], [10] but never deployed on a large-scale overlay.

The *NRTT* module is an implementation of the NICE module plus low-RTT bias. While NICE follows Kademlia's rules regarding node replacement —i.e. nodes cannot be replaced

unless they fail to respond to queries— NRTT introduces the possibility of replacing an existing node with a recently discovered node, if the RTT of the incoming node is lower than that of the existing node.

D. NRTT + 128-bucket (NR128)

Another approach to improve performance is to reduce the number of lookup hops. The most extensive study of bucket modifications in Kademlia [3] considered two options: (1) adding more buckets to the routing table and (2) enlarging existing buckets. Their theoretical analysis concluded that, while both approaches offer comparable hop reduction on average, increasing bucket size is simpler to implement, has lower maintenance cost, and improves resistance to churn as a side effect. Finally, they showed that performance improves logarithmically with bucket size.

Enlarging buckets is simple but costly because maintenance traffic grows linearly with bucket size. That is, if one is to enlarge all buckets equally. But not all the buckets are equal when it comes to lookup performance.

Given the structure of a Kademlia routing table, on average, the first bucket is used in half of the lookups, the second bucket in a quarter of the lookups, and so forth. In the *NR128* routing module, buckets are enlarged proportionally to the probability of them being used in a given lookup. The first buckets hold 128, 64, 32, and 16 nodes respectively, while the rest of the bucket sizes remain at 8 nodes. To our knowledge, this technique has not been proposed before.

The expected result is that, while half of the lookups are bootstrapped by a 128-bucket, and more than nine in ten by an enlarged bucket, maintenance traffic merely doubles compared to NICE (20 queries per minute).

VI. Experimental Setup

To measure and understand the behavior of UTorrent and of our own implementations, we have run numerous experiments in a variety of configurations, both sequential and parallel. Our final configuration is one in which we ran all implementations in parallel, providing the same experimental conditions to all nodes being compared, on a large number of freshly acquired torrent *infohashes* (see below). The experiment we document in this paper is neither the best nor the worst but rather representative, as the results we obtained are very consistent between different runs.

The experiment, which started on March 26, 2011 and ran over 80 hours, tested all our eight (two times four) implementations and UTorrent version 2.2 (build 23703)[3]. A very simple coordination script is used to command our nodes under test; a Python interface is used for our MDHT node implementations and an HTTP interface is used for UTorrent.

Each node under test joins the multimillion-node MDHT overlay. Upon joining the overlay, the lookup rounds begin. In each round, a random NuT sequence is generated. Every 10 seconds, the next NuT in the sequence is commanded to perform a lookup on an identifier that is randomly selected from its list of infohashes and then remove the infohash from its list. All nine NuTs perform one lookup per round, until every NuT has emptied its list of infohashes —i.e. when every NuT has performed a lookup for every infohash. The experiment is then considered complete and the captured traffic is ready to be parsed by our profiling toolkit.

[3]Downloaded from http://www.utorrent.com/downloads/

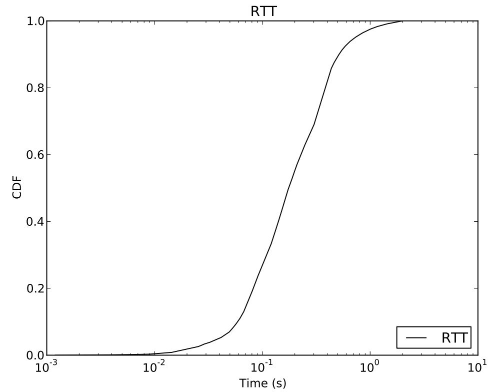

Fig. 2. RTT to nodes in the MDHT overlay (Queries that have not been replied within 2 seconds are considered timed-out, thus excluded from this graph.)

Experiments were not CPU-bound, and were run on a system with a P4@3.0GHz, 3GB RAM and running Windows 7. Regarding network latency, as shown in Figure 2, the RTTs from the nodes under test to other MDHT nodes were mainly concentrated between 100 and 300 ms, with very few RTTs over one second (2nd percentile: 2.13 ms, 25th percentile: 94.8 ms, median: 175.2 ms, 75th percentile: 343.6 ms, 98th percentile: 1093.9 ms).

Infohashes can be obtained from various sources, and can even be generated by us. We are, however, specifically interested in active swarms under "real world" conditions. This has led us to obtain infohashes from one of the most popular BitTorrent sites on the Internet, `thepiratebay.org`. This site has a "top" page with the most popular content organized in categories. We have extracted all infohashes from these categories, obtaining a total of 3078 infohashes.

It should be noted that our MDHT node implementations neither download, nor offer for upload, any content associated with these infohashes. UTorrent is given only 3 seconds to initiate the download —triggering a DHT lookup as a side-effect— before being instructed to stop its download, thus leaving no time for any data transfer. We have observed that the DHT lookup progresses normally despite the stop command.

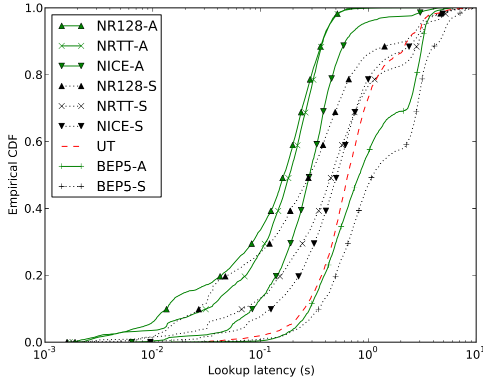

Fig. 3. Lookup latency when retrieving a value from the MDHT overlay

VII. Experimental Results

Each of the eight node implementations we have studied is one of the combinations of our four routing and two lookup modules described previously, and the names we use for them reflect the components combined. For instance, the NICE-S node implementation uses the NICE routing module plus the standard (S) lookup module. Similarly, we will use "wildcards". For example, *-S indicates all nodes using the standard lookup module, and NICE-* all nodes using the NICE routing module. UTorrent is simply referred to as UT.

A. Lookup Latency

Figure 3 shows the empirical cumulative distribution function (CDF) of lookup latency when retrieving a value from the overlay, as defined in Section III-B.

In Table I we document lookup latencies for the 50 (median), 75, 98, and 99 percentiles. We expect these figures to be valuable for those interested in large-scale latency-sensitive systems, whose requirements usually specify a maximum latency for a large fraction of the operations.

Since our BEP5 routing module follows the MDHT specifications published by the creators of UTorrent (BitTorrent, Inc.) and our standard module implements the same lookup parameters as UTorrent, we expected that BEP5-S would perform similarly to UTorrent. As our measurements reveal, this is not the case. UT performs significantly better than our BEP5-S, and even outperforms our BEP5-A, which we expected to beat UT by using more aggressive lookups. This fact suggests undocumented enhancements in UTorrent's routing table management.

We see that the aggressive lookup module consistently yielded lower median lookup latency, but more importantly, drastically reduced the worst-case latencies, as seen in the 98th and 99th percentile columns of Table I.

Nodes implementing the NICE routing module perform better than BEP5 and also UTorrent. We believe that this is due to an improvement in the quality of the initiator's routing table caused by our constant refresh strategy and mechanisms to detect and avoid nodes with connectivity limitations.

TABLE I
Lookup Latency (in ms)

Node	median	75th p.	98th p.	99th p.
UT	647	1047	3736	5140
BEP5-S	1105	3011	6828	7540
NICE-S	510	877	4468	5488
NRTT-S	459	928	5060	5737
NR128-S	286	589	4375	5343
BEP5-A	825	2601	3840	4168
NICE-A	284	420	2619	3247
NRTT-A	185	291	512	566
NR128-A	164	269	506	566

The performance gain from the addition of low-RTT bias (NICE vs. NRTT) is uneven. NRTT-A performs significantly better than NICE-A, but using standard lookups, the difference is less pronounced. This is due to standard lookups not being able to take full advantage of low-RTT contacts by rapidly fanning out. The comparison between NRTT-S and NRTT-A illustrates this point, where the worst-case latency is an order of magnitude lower for NRTT-A.

When examining the routing table, we find that NICE-* nodes have contacts with RTTs in the 100–300 ms range, while NRTT-* nodes have contacts whose RTTs are lower than 20 ms.

Conversely, we see that the impact of enlarged routing tables in NR128-variants is the opposite to that in NRTT. The small performance gain from NRTT-A to NR128-A may be a sign that the maximum performance has been reached already. Indeed, NR128-A's median lookup latency is, in fact, lower than the median RTT to MDHT nodes.

NR128-A, our best performing node implementation, achieves a median lookup latency of 164 ms. While median lookup latency is important, many latency-sensitive applications are more concerned with the worst-case performance, and treat lookup latency above a narrow threshold as failure.

Where previous measurements of large-scale Kademlia-based overlays report long tails with worst-case latencies in the tens of seconds, our NRTT-A and NR128-A consistently achieve sub-second lookups, with almost 98% finishing in less than 500 ms. More importantly, assuming a hard lookup deadline of 1 second, less than five out of over three thousand lookups would fail using any of these two implementations.

B. Lookup Cost

Lookup cost, defined in Section III-B, is also an important characteristic to measure. As Figure 4 shows, implementations using the aggressive lookup module require more lookup queries, thus increasing the lookup cost.

Lookup cost in UTorrent and our *-S nodes are very similar, as we expected. Among them, NRTT-S is slightly more expensive than the rest, which is caused by a more intensive query burst, due to the fan-out effect discussed in the previous section. Conversely, NR128-S has the lowest lookup

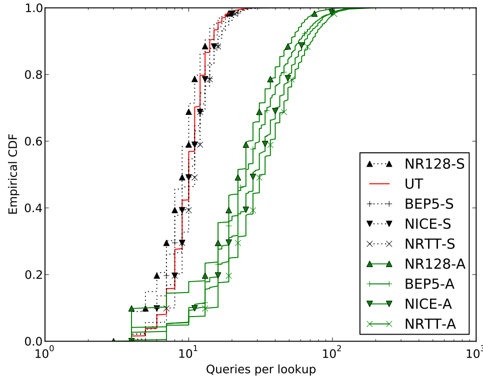

Fig. 4. Lookup cost. (We recognize that distinguishing between individual lines in this graph is hard, but the difference between standard (*-S and UT) and aggressive (*-A) lookup is clear. Also notice the tendency of NR128-* and NRTT-* to have lower and higher cost than the rest, respectively.)

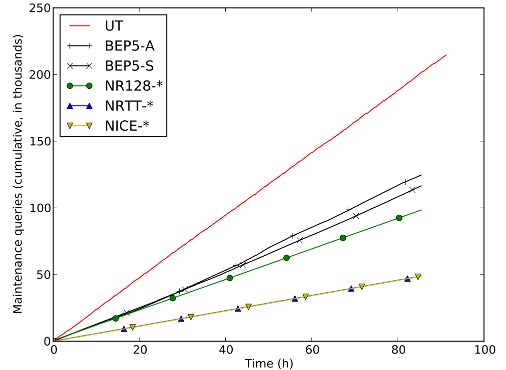

Fig. 5. Cumulative maintenance traffic during the entire experiment

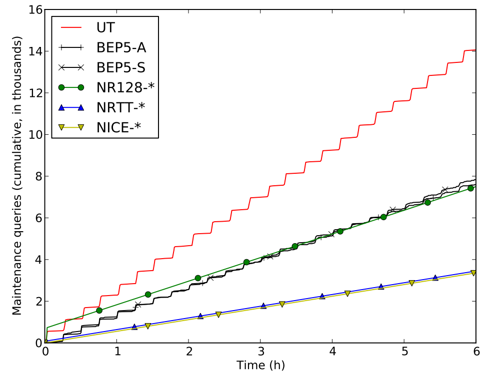

Fig. 6. Cumulative maintenance traffic during the first 6 hours

cost, which comes as no surprise since its enlarged buckets will reduce the number of hops required.

We predictably see similar relationships between the *-A nodes, but with higher average lookup costs across the board.

C. Maintenance Cost

Figure 5 depicts the cumulative maintenance queries sent over time. The obtained results confirm that all our MDHT node implementations generate less maintenance traffic, by a considerable margin, than UTorrent.

As mentioned earlier, we believe that UTorrent's unexpectedly good lookup performance is due to modifications to its routing table management, compared to the specification. This would go a long way towards explaining why we observe much more maintenance traffic than for our BEP5-* implementations.

Figure 6 shows only the first 6 hours of the experiment, revealing a peculiar stair-like pattern in UT and BEP5-*. Every 15 minutes, UTorrent triggers a burst of maintenance messages, approximately 600 messages for a period of 1–2 minutes, and few or no queries between bursts. We see a similar pattern initially in our own BEP5-* implementations, but they quickly flatten out. This observation suggests that while the initial occurrence is an artifact of the specification, the continued behavior is due to the way UTorrent implements its internal synchronization mechanism, causing maintenance message bursts.

All our MDHT node implementations, regardless of the modifications they include, drastically reduce maintenance traffic compared to UTorrent. BEP5-S and BEP5-A have irregular maintenance traffic patterns while the rest were designed to have very regular traffic patterns.

The enlarged bucket implementations (NR128-*) generate twice the maintenance traffic of NICE-* and NRTT-* (whose lines overlap), but still generate less traffic than BEP5-* in the long run.

D. Trade-offs

In comparing our different implementations, we have explored different trade-offs between performance and cost. We do, however, also see that some benefits can be gained at zero, or even negative, cost. For instance, NR128-S is better than UTorrent in all aspects, with significantly lower maintenance cost, lower lookup cost and median lookup latencies less than 50% of UTorrent's. Both NR128-S and UTorrent suffer from long tails, however, with 10% and 14%, respectively, of the lookups taking more than 2 seconds.

While achieving better performance at lower cost is certainly desirable, our target applications have very strict latency requirements. We are thus forced to go a step further, and

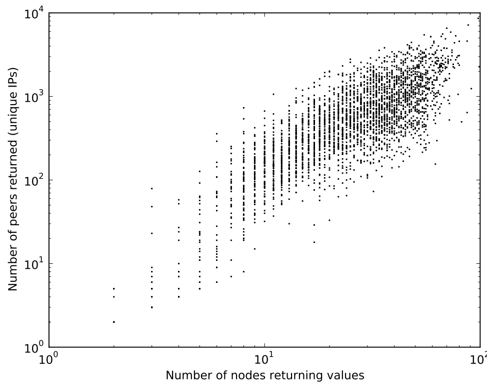

Fig. 7. Peers versus nodes returning values

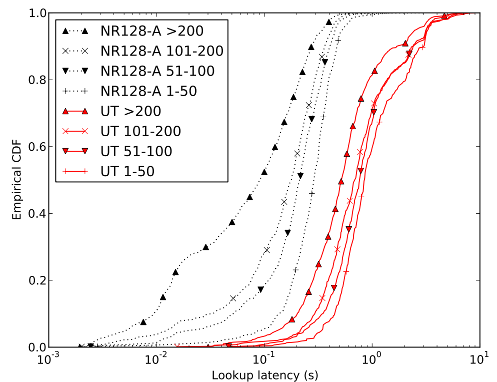

Fig. 8. Lookup latency versus swarm size for UTorrent and NR128-A. Both implementations perform better on larger swarms.

carefully explore what trade-offs we can make to meet these requirements, even at sometimes significantly higher costs.

Specifically, our low-RTT bias nodes (NRTT-*) achieve a noticeable performance improvement while keeping the same maintenance cost and just a small increase in lookup cost, which we attribute to being able to more rapidly fan out queries, compared to NICE-*. Finally, enlarging buckets improves lookup performance while slightly reducing lookup cost, which might be suitable when lookup cost dominates over maintenance cost. For example, a system where lookups are performed very frequently.

VIII. Additional Results

Our primary goal was to reduce the time until our node under test received the first value, but since we have not changed the way lookups terminate, they will continue until they reach the node closest to the key. Our toolkit continued to capture information about this phase of the lookups as well.

The modifications we have made have implications not only for the first phase, analyzed in the previous section, but for the complete lookup. In this section, we will present our analysis and summarize our results as they apply to the whole lookup.

A. Lookup Latency Versus Swarm Size

In principle, in a Kademlia-based DHT, only a fixed number of nodes need to store the values corresponding to a given key, regardless of the size of DHT or the popularity of the key. In practice, we find that popular keys in MDHT tend to have values distributed among a large number of nodes, while less popular keys are less widely dispersed.

In Figure 7 we plot the number of nodes returning values (replicas) against number of unique values stored (swarm size). We see that as swarm size increases, the number of replicas found increases as well.

This has no impact on the time it takes to reach the node closest to the key, but has a significant impact on lookup latency, as defined by us. Not only because there are more replicas to find, but also because having more replicas increase the chance that at least one of them is close (low RTT) to the node under test.

We thus see a relationship between swarm size and lookup latency. Figure 8 illustrates that lookup latency is lower when looking up popular infohashes (large swarms). In NR128-A, for instance, median lookup latency for swarms with more than 200 peers is 92 ms versus 289 ms for swarms with 50 peers or less (521 ms vs. 848 ms in UTorrent).

We draw two conclusions from these observations. First, users should expect significantly lower latency when looking up popular keys (i.e., popular content). And second, our techniques yield medians well under half second even for small swarms.

B. Reaching the Closest Node

In this paper, we have focused on a more user-centric metric of DHT performance, the time to find values. Another metric that has been widely used and studied in DHTs is the time to reach the closest node to the target key [3]. For completeness, and to allow our results to be easily compared to previous work, we plot our results according to this metric in Figure 9.

Using this metric, our NR128-A implementation still achieves sub-second results, with a median of 455 ms and 92.8% of its lookups reaching the closest node within a second.

C. Queries & Responses

The Internet is a pretty hostile environment, and many issues that normally would not arise in a testbed or simulator will impede performance when the same code is deployed “in the wild”. As an example, Table II presents information about lookup traffic obtained in our experiments. The *queries* columns show the number of queries generated, and *responses* the responses received, both the absolute number and as

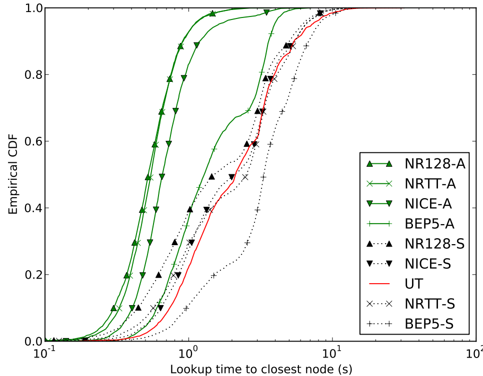

Fig. 9. Lookup latency to reach the closest node to the key

TABLE II
LOOKUP QUERIES AND RESPONSES

Label	Queries	Responses (%)
UT	92,450	52,378 (57)
BEP5-S	67,454	36,361 (54)
NICE-S	68,923	43,937 (64)
NRTT-S	70,234	44,515 (63)
NR128-S	64,488	39,633 (61)
BEP5-A	198,015	116,070 (59)
NICE-A	260,849	166,026 (64)
NRTT-A	281,335	183,175 (65)
NR128-A	221,543	140,025 (63)

TABLE III
IMPLEMENTATION MARKET SHARE

Implementation	Nodes (unique IPs)	Percentage
UT	2,663,538	60.0
LT	324,122	7.3
TR	7,666	0.2
Other versions	4,813	0.1
No version	1,441,899	32.5
Total	4,442,038	100.0

a percentage of queries issued, with the remainder being timeouts.

As can be seen, for BEP5-* and UT, less than 60% of queries receive responses, the equivalent, one could say, of more than a 40% packet loss ratio. In previous work [16], we characterized these connectivity artifacts and proposed mechanisms to identify and filter out nodes with connectivity issues. In this paper, some of these mechanisms have been implemented, improving the quality of our own routing tables. We believe that these improvements explain why NICE-*, NRTT-* and NR128-* consistently see a higher response rate than BEP5-* and UT. We plan to analyze the impact of these routing policies on the quality of routing tables in future work.

D. Implementation Market Share

Mainline DHT messages have an optional field where the sender can indicate its version in a four-character string. The first two characters indicate the client —UTorrent nodes identify themselves as “UT”— and the other two, the version number. The client labels reported by nodes are presented in Table III.

During the course of this experiment, the nodes under test have exchanged messages with over four million nodes (unique IP addresses) in the MDHT overlay. We have identified 2.6 out of 4.4 million (60%) as UTorrent nodes, far ahead of the second most common node implementation, libtorrent. It is also noteworthy that about one third of the nodes did not include this optional field in their messages.

IX. Related Work

Li et al. [18] simulated several DHTs under intensive churn and lookup workloads, in order to understand and compare the effects of different design properties and parameter values on performance and cost. The study revealed that, under intensive churn, Kademlia’s capacity of performing parallel lookups reduces the effect of timeouts compared to other DHT designs studied. In their simulation results, Kademlia achieved a median lookup latency of 450 ms with the best parameter settings.

Kaune et al. [10] proposed a routing table with a bias towards geographically close nodes, called *proximity neighbour selection (PNS)*. Although their goal was to reduce inter-ISP traffic in Kademlia, they observed that PNS also reduced lookup latency in their simulations from 800 to 250 ms.

Other non-Kademlia-based systems have been studied. Rhea et al. [19] showed that an overlay deployed on 300 PlanetLab hosts can achieve low lookup latencies (median under 200 ms and 99th percentile under 400 ms). Dabek et al. [20] achieved median lookup latencies between 100–300 ms on an overlay with 180 test-bed hosts.

Crosby and Wallach [7] measured lookup performance in two Kademlia-based large-scale overlays on the Internet, reporting a median lookup latency of around one minute in Mainline DHT and two minutes in Azureus DHT. They argue that one of the causes of such performance is the existence of dead nodes (non-responding nodes) in routing tables combined with very long timeouts. Falkner et al. [8] reduced ADHT’s median lookup latency from 127 to 13 seconds by increasing the lookup cost three-fold.

Stutzbach and Rejaie [3] modified eMule’s implementation of KAD to increase lookup parallelism. Their experiments revealed that lookup cost increased considerably while lookup latency improved only slightly. Their best median lookup latency was around 2 seconds.

Steiner et al. [4] also tried to improve lookup performance by modifying eMule’s lookup parameters. Although they discovered that eMule’s design limited their modifications’ impact, they achieved median lookup latencies of 1.5 seconds on the KAD overlay.

X. Conclusion

In this paper, we have shown that it is possible for a node participating in a multimillion-node Kademlia-based overlay to consistently perform sub-second lookups. We have also analyzed the impact of each proposed modification on performance, lookup cost, and maintenance cost, exposing the trade-offs involved. Additionally, we observed a phenomenon relevant for applications using the overlay: the more popular a key is, the faster the lookup.

In our efforts to accomplish the goal of supporting latency-sensitive applications using Mainline DHT, we have also produced other noteworthy secondary results, including, but not limited to: **(1)** a profiling toolkit that allows us to analyze MDHT messages exchanged between the node under study and other MDHT nodes, without code instrumentation; **(2)** the deployment and measurement of three modifications to routing table management (NICE, NRTT, NR128); and **(3)** an infrastructure to rapidly implement and deploy those modifications in the form of plug-ins.

Our initial study of MDHT node implementations revealed that UTorrent is the most common implementation currently in use, with a measured "market share" of 60%, making UTorrent a good candidate as the state-of-the-art benchmark for us to beat.

Our most aggressive implementation (NR128-A) not only beats UTorrent, but also steals its lunch money. Not only is our median lookup latency almost four times lower than UTorrent's, but, most importantly for our purposes, just 0.1% of NR128-A's lookups need over a second versus over 27% of UTorrent's. While this comes at a higher lookup cost (220%), when we consider both lookup and maintenance traffic, our implementation actually generates substantially less traffic than UTorrent.

Amongst our less aggressive lookup implementations, NR128-S needs slightly less queries per lookup, half the maintenance traffic, and still its median lookup latency is less than half of UTorrent's, beating it in all three metrics.

We hope that others will find our results useful in designing, evaluating, and improving applications deployed on top of large-scale DHT overlays on the Internet. All the source code described in this paper is available on-line at: http://people.kth.se/~rauljc/p2p11/.

Acknowledgment

The authors would like to thank Rebecca Hincks, Amir H. Payberah, and the anonymous reviewers for their valuable comments on our drafts.

The research leading to these results has received funding from the Seventh Framework Programme (FP7/2007-2013) under grant agreement No. 216217 (P2P-Next).

References

[1] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the XOR metric," in *Proceedings of the 1st International Workshop on Peer-to Peer Systems (IPTPS02)*, 2002, pp. 53–65.

[2] B. Cohen, "Incentives Build Robustness in BitTorrent," in *Workshop on Economics of Peer-to-Peer Systems*, vol. 6. Berkeley, CA, USA, 2003.

[3] D. Stutzbach and R. Rejaie, "Improving Lookup Performance Over a Widely-Deployed DHT," in *INFOCOM*. IEEE, 2006.

[4] M. Steiner, D. Carra, and E. W. Biersack, "Evaluating and improving the content access in KAD," *Springer "Journal of Peer-to-Peer Networks and Applications", Vol 2*, 2009.

[5] M. Steiner, T. En-Najjary, and E. W. Biersack, "A global view of kad," in *IMC '07: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. New York, NY, USA: ACM, 2007, pp. 117–122.

[6] K. Junemann, P. Andelfinger, J. Dinger, and H. Hartenstein, "BitMON: A Tool for Automated Monitoring of the BitTorrent DHT," in *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on*. IEEE, 2010, pp. 1–2.

[7] S. A. Crosby and D. S. Wallach, "An analysis of bittorrent's two kademlia-based dhts," 2007.

[8] J. Falkner, M. Piatek, J. P. John, A. Krishnamurthy, and T. Anderson, "Profiling a million user DHT," in *IMC '07: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. New York, NY, USA: ACM, 2007, pp. 129–134.

[9] J. Li, J. Stribling, T. M. Gil, R. Morris, and M. F. Kaashoek, "Comparing the performance of distributed hash tables under churn," in *In Proc. IPTPS*, 2004.

[10] S. Kaune, T. Lauinger, A. Kovacevic, and K. Pussep, "Embracing the peer next door: Proximity in kademlia," in *Eighth International Conference on Peer-to-Peer Computing (P2P'08)*, 2008, p. 343–350.

[11] A. Bakker, R. Petrocco, M. Dale, J. Gerber, V. Grishchenko, D. Rabaioli, and J. Pouwelse, "Online video using bittorrent and html5 applied to wikipedia," in *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on*, 8 2010, pp. 1 –2.

[12] M. J. Freedman, "Experiences with coralcdn: a five-year operational view," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, ser. NSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 7–7. [Online]. Available: http://portal.acm.org/citation.cfm?id=1855711.1855718

[13] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz, "Tapestry: a resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 41–53, 2004.

[14] A. Rowstron and P. Druschel, "P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," *In: Middleware*, pp. 329–350, 2001.

[15] A. Loewenstern, "BitTorrent Enhancement Proposal 5 (BEP5): DHT Protocol," 2008.

[16] R. Jimenez, F. Osmani, and B. Knutsson, "Connectivity properties of Mainline BitTorrent DHT nodes," in *9th International Conference on Peer-to-Peer Computing 2009*, Seattle, Washington, USA, 9 2009.

[17] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica, "The impact of DHT routing geometry on resilience and proximity," in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, 2003, pp. 381–394.

[18] J. Li, J. Stribling, R. Morris, M. F. Kaashoek, and T. M. Gil, "A performance vs. cost framework for evaluating DHT design tradeoffs under churn," in *INFOCOM*, 2005, pp. 225–236.

[19] S. Rhea, B. Chun, J. Kubiatowicz, and S. Shenker, "Fixing the embarrassing slowness of OpenDHT on PlanetLab," in *Proc. of the Second USENIX Workshop on Real, Large Distributed Systems*, 2005, pp. 25–30.

[20] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris, "Designing a dht for low latency and high throughput," in *IN PROCEEDINGS OF THE 1ST NSDI*, 2004, pp. 85–98.