ROYAL INSTITUTE
OF TECHNOLOGY

Distributed Peer Discovery in Large-Scale P2P Streaming Systems

Addressing Practical Problems of P2P Deployments on the Open Internet

RAUL JIMENEZ

Doctoral Thesis in Communication Systems
Stockholm, Sweden 2013

TRITA-ICT/ECS AVH 13:19
ISSN 1653-6363
ISRN KTH/ICT/ECS/AVH-13/19-SE
ISBN 978-91-7501-917-8

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 onsdag den 11 december 2013 klockan 13:00 i Ka-Aula (Aulan), Kungl Tekniska högskolan, Forum, Isafjordsgatan 39, Kista.

Tryck: Universitetsservice US AB

Abstract

Peer-to-peer (P2P) techniques allow users with limited resources to distribute content to a potentially large audience by turning passive clients into *peers*. Peers can self-organize to distribute content to each other, increasing the scalability of the system and decreasing the publisher's costs, compared to a publisher distributing the data himself using a content delivery network (CDN) or his own servers.

Peer discovery is the mechanism that peers use to find each other. Peer discovery is a critical component of any P2P-based system, because P2P networks are dynamic by nature. That is, peers constantly join and leave the network and each individual peer is assumed to be unreliable.

This thesis addresses practical issues in distributed peer discovery mechanisms in the context of three different large-scale P2P streaming systems: a (1) BitTorrent-based streaming system, (2) Spotify, and (3) our own mobile P2P streaming system based on the upcoming Peer-to-peer Streaming Protocol (PPSP) Internet standard.

We dramatically improve peer discovery performance in BitTorrent's Mainline DHT, the largest distributed hash table (DHT) overlay on the open Internet. Our implementation's median lookup latency is an order of magnitude lower than the best performing measurement reported in the literature and does not exhibit a long tail of high-latency lookups, which is critical for P2P streaming applications.

We have achieved these results by studying how connectivity artifacts on the underlying network —probably caused by network address translation (NAT) gateways— affect the DHT overlay. Our measurements of more than three million nodes reveal that connectivity artifacts are widespread and can severely degrade DHT performance.

This thesis also addresses the practical issues of integrating mobile devices into P2P streaming systems. In particular, we enable P2P on Spotify's Android app, study how distributed peer discovery affects energy consumption, and implement and evaluate backwards-compatible modifications which dramatically reduce energy consumption on 3G.

Then, we build the first complete system that not only is capable of streaming content to mobile devices but also allows them to publish content directly into the P2P system, even when they are behind a NAT gateway, with minimal impact on their battery and data usage.

While our preferred approach is implementing backwards-compatible modifications, we also propose and analyze backwards-incompatible ones. The former allow us to evaluate them in the existing large-scale systems and allow developers to deploy our modifications into the actual system. The latter free us to propose deeper changes. In particular, we propose (1) a DHT-based peer discovery mechanism that improves scalability and introduces locality-awareness, and (2) modifications on Spotify's gossip-like peer discovery to better accommodate mobile devices.

To my wife Svetlana,
our daughter Laura,
and our soon-to-be-born son

Acknowledgements

This thesis is the result of my work in collaboration with my paper's co-authors: Björn Knutsson Flutra Osmani, Seif Haridi, Victor Grishchenko, Gunnar Kreitz, Marcus Isaksson, Arno Bakker, and Johan Pouwelse. While I claim most of the contribution as my own, this thesis could not have been possible without their hard work and support. Thus, 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.

I am grateful to Björn Pehrson, my advisor during my first years, who is so passionate about his work that is hard to notice that he has already retired. I thank Seif Haridi, who took his role as advisor when Björn Pehrson retired. He has been very supportive and I have enjoyed spending time with him and his research group at the Swedish Institute of Computer Science (SICS).

Many thanks to all my colleagues at Network Systems Laboratory (NSLab), with whom I have had interesting discussions about my research and a great deal of fun. In particular, Flutra Osmani deserves a good deal of gratitude for her collaboration in an important part of the research in this thesis.

I must thank all people I interacted with during my seven-month research visit at Spotify. Gunnar Kreitz and Marcus Isaksson were of great help in my research. Javier Ubillos, Pablo Barrera, Guido Urdaneta, and Tommie Gannert made me feel at home and helped in different ways: from discussing ideas to soldering wires to a smartphone's battery.

Thanks to my colleagues in the P2P-Next project. It is great to collaborate with such smart and friendly people. In particular, our collaboration with Johan Pouwelse and Victor Grishchenko of TU Delft produced three papers (two of them included in this thesis). Lars-Erik of Dacc collaborated in another paper, although that paper is not included in this thesis.

I sincerely appreciate the effort of the following people to make room in their busy agendas to evaluate this thesis and provide valuable feedback. Arnaud Legout of INRIA (opponent), Jim Dowling of SICS (internal reviewer), and the examination committee members: Peter Van Roy of Catholic University of Louvain, Luigi Liquori of INRIA, and Rolf Stadler of KTH.

Finally, I thank my family and friends for their unconditional support, especially my wife Svetlana Panevina. Our daughter's smile and the expectation of our soon-to-be-born son have greatly helped to counter these dark periods one goes through while writing a thesis.

Funding

The research presented in this thesis has received funding from:

- the Seventh Framework Programme under grant agreement No. 216217 (P2P-Next project). Web site: `http://p2p-next.org/`
- the Swedish Foundation for Strategic Research (E2E-Clouds project). Web site: `http://e2e-clouds.org/`
- EIT ICTLabs activity 12199 (Seamless P2P video streaming for the web). Web site: `http://www.ppsp.me/`

Errata

In papers [2], [3] and [4] (acknowledge section), the project number should be 216217 instead of 21617. Papers [2] and [3] are reproduced in their original form in Chapters 6 and 7, respectively.

Contents

Part I

Thesis Overview

Chapter 1

Introduction

Over the years, P2P techniques have been used to distribute content to millions without the necessity of large resources on the part of the content provider. This has been possible because end-users (called *peers*) not only receive data but also contribute to its distribution with their own resources.

Peer discovery is the mechanism allowing any peer to find other peers interested in the same piece of content. Once a peer finds other peers, it can establish direct connections and download the content from a subset of these peers. Then, this peer can be found by other peers, which in turn, will download data from this peer.

This thesis studies two existing large-scale distributed peer discovery systems through a series of experiments: BitTorrent's Mainline DHT (a tracker-like implementation over a distributed hash table overlay) and Spotify's gossip-like overlay network. First, we identify issues hindering the adaptation of these systems to new contexts (e.g., latency-critical lookups for multimedia streaming and integration of mobile devices). Then, we propose modifications to these systems to adapt them to the new requirements demanded by new features and changes in the underlying infrastructure (battery-powered devices, NATs and firewalls).

In addition to these two existing large-scale P2P systems, this thesis introduces two systems designed to be deployed at large scale on the open Internet: CTracker and Tribler Mobile. The first one is a topology-aware DHT-based peer discovery mechanism that provides topologically close peers with the final goal of reducing inter-ISP traffic. The second one is the adaptation of an existing P2P protocol to allow smartphones to not only play P2P video streams but also broadcast them, considering important challenges such as connectivity artifacts (NAT gateways and firewalls) and energy consumption.

Tribler Mobile is based on Swift, the protocol chosen to become part of the upcoming Internet standard Peer-to-Peer Streaming Protocol (PPSP). This thesis contains a paper where we argue that Swift's properties largely fulfill Information Centric Network's (ICN) requirements, thus enabling an incremental deployment of ICN applications on the existing Internet infrastructure. Although addressing

ICN issues is not an explicit goal in this thesis, I include this paper because it helps to understand Swift's properties.

1.1 Low-Latency DHT on the Open Internet

The first major goal of this thesis is to understand and close the substantial performance gap between DHT overlay networks in the lab, where latencies are typically well below one second, and large-scale DHT overlay networks on the open Internet, where latencies are typically measured in seconds or even tens of seconds. We tackle this problem with two different approaches: a backwards-compatible one which focuses solely on DHT lookup performance and a backwards-incompatible one that not only aims to improve DHT lookup performance but also introduce topology-awareness in the P2P system.

Our first approach is to study BitTorrent's Mainline DHT (MDHT). MDHT is the largest DHT overlay on the Internet [5], and its lookup latencies have been measured in seconds in the literature [6]. Then, we propose backwards-compatible modifications[1]. Finally, we implement, deploy, and evaluate these modifications by running extensive experiments on the actual multi-million-node DHT overlay.

Our backwards-compatible modifications do close the performance gap, enabling latency-sensitive services on large-scale DHT-based systems on the Internet. In particular, our code is used by NextShare, a fully-distributed content streaming platform whose responsiveness should be able to compete with cable and satellite TV latencies. This platform has been developed by the P2P-Next project[2]

Our second approach is to propose a different DHT-based peer discovery mechanism for BitTorrent which addresses MDHT's scalability issues and introduces topology-awareness.

The following sections elaborate on technical details and contributions.

1.1.1 Connectivity Properties

Previous research by Crosby and Wallach [6] indicated that connectivity artifacts on the underlying network may be the main cause of the poor performance in MDHT. In their study, they found that a large fraction of the queries sent during a lookup are never responded to, causing long delays. While others have proposed approaches to address this issue (e.g., reducing, or removing, timeout delays [7]), 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 open Internet in our case). In fact, few overlay algorithms explic-

[1] To evaluate the modifications on the actual DHT overlay, they must be backwards-compatible with existing DHT nodes.

[2] http://www.p2p-next.org/ (Nov. 2013)

itly state their underlying connectivity assumptions. Synapse is one such example: "To ensure the usual properties of the underlying network, we [Synapse's authors] assume that communication is both symmetric and transitive" [8].

In Chapter 7, we characterize nodes' connectivity using three connectivity properties: **reciprocity**, **transitivity**, and **persistence**. We define each of these properties clearly and we provide a mechanism to measure the connectivity properties of any node in the Mainline DHT overlay.

Overlays rely on networking infrastructure (called *underlay networks*) being able to deliver messages from one node to another. In an ideal underlay network, the connectivity between any two nodes would exhibit all three properties.

The open Internet, however, is far from being an ideal underlay network. Our measurements of over 3.6 million Mainline DHT nodes on the open Internet, reveal a large fraction of these nodes lacking one or more connectivity properties.

We study the impact of the lack of each of these properties on the DHT overlay, concluding that these connectivity artifacts degrade performance. This is because DHT designs *implicitly assume* that the underlying network provides all three connectivity properties, but that assumption does not hold on today's open Internet[3].

In previous work, Freedman et al. [9] studied non-transitive connectivity caused by a network partition on PlanetLab [10] and how it degraded DHT performance. NAT gateways, however, cause subtler connectivity artifacts, which we study using the three connectivity properties defined above.

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

Chapter 7 describes the mechanisms we propose. A brief account of the deployment and evaluation of some mechanisms is presented next.

1.1.2 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 [11] and CoralCDN [12], 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 autonomous 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 backwards-

[3]To be fair, we must say that it was not trivial for these designers to foresee the massive deployment of NAT gateways on the Internet during the last years.

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 global lookup performance in the DHT overlay, we try to avoid global modifications whose deployment is slower and more complicated. Instead, we take a bottom-up approach, where we start with local backwards-compatible modifications that mainly improve lookup performance at the node where modifications are implemented. 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 interoperable (i.e., backwards compatible) with existing node implementations.

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 backwards 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 lookup performance (i.e., the lookup performance of the node we modify), 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 are expected to benefit existing nodes as well. Thus, as developers include our modifications in their clients, all nodes benefit. We elaborate on these local and global benefits in Chapter 8.

1.1.3 Scalability and Locality-Awareness

This thesis also considers adding locality-aware features to the DHT, which would facilitate BitTorrent peers in finding 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 [13] studied this problem and proposed their own solution, which unlike ours, assumes that content popularity is known.

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. [14] proposed a different mechanism where ISPs inject locality-aware nodes in the Mainline DHT overlay.

In Chapter 6, we describe our design and provide technical details of Tapestry [15], the DHT design on which we base our design.

1.2 Integrating Smartphones into P2P Systems

The second major goal of this thesis is to identify and address challenges related to the integration of smartphones into P2P systems. In particular, we study the

involvement of distributed peer discovery mechanisms in these challenges and propose modifications to these distributed peer discovery mechanisms to adapt them to mobile devices.

While P2P-based file-sharing applications are still very popular on desktop devices (PCs and laptops), the usage of P2P-based systems on mobile devices (smartphones and tablets) remains negligible. It is not that smartphone users do not consume multimedia content. Mobile multimedia consumption is a significant part of the total and is growing at faster rate than consumption on desktop devices. Although, mobile consume content mainly from cloud-based services such as YouTube and content distribution networks (CDNs). For instance, mobile devices consumed 41% of YouTube's total traffic in the third quarter of 2013, up from 25% in 2012 and just 6% in 2011.[4]

Even Spotify [16, 17] —a commercial streaming service whose desktop clients stream about 80% of the data via P2P— turns off P2P distribution on mobile devices, letting these devices stream data from their servers or CDNs, instead.

Energy consumption and current data caps on mobile networks (usually monthly allowances of 0.5 to 2 GB) are often considered the main reasons why users are reluctant to use P2P-based apps and why companies do not enable P2P on their apps.

Certainly, a mobile peer uploading content to others would consume more energy and data than its client-server counterpart because in the client-server system clients do not upload content at all.[5] But consider now that the mobile peer can download from other peers and never upload to others. That is, the mobile peer behaves like —in BitTorrent's terminology— a *leecher*.[6]

Mobile Leechers

In this thesis, we propose, implement, and evaluate two different P2P streaming systems where peers running on mobile devices are *leechers*.

We use the term *leecher* for mobile devices that download content from other peers but do not upload to others. Although the term leecher is commonly used pejoratively for free-riders in P2P file-sharing communities, we use it without any negative connotation. On the contrary, our goal is to address the extra capacity demanded by leechers by offsetting it with extra capacity provided by *seeders*. These seeders are desktop peers, which can be supplemented by peers run by the content provider to improve users' quality of experience.

[4] `http://www.computerworld.com/s/article/9243331/` (Nov. 2013)

[5] Assuming that the server can serve content as fast as the P2P network. If downloading from a server took longer time than downloading from peers, it could happen that a client-server download would consume more energy than the *full peer* alternative.

[6] Nurminen and Noyranen [18] observed that uploading can contribute to shorter download time, and thus lower total energy consumption in BitTorrent due to BitTorrent's tit-for-tat mechanism [19]. While we are aware of this effect, we do not consider it a relevant factor in our work because the systems we study here heavily rely on seeding, thus diminishing the impact of the tit-for-tat incentive mechanism.

Note that our proposal does not discard the possibility of addressing the above-mentioned challenges of integrating mobile devices as full peers. We deliberately choose to limit our scope to *mobile leechers* to provide a working solution now and leave the door open to incremental steps towards *full mobile peers* in the future.

Leaving upload-related issues aside allows us to focus on the distributed peer discovery mechanisms and how they function in a P2P system where mobile devices are integrated as leechers.

The following two sections introduce our work on two different systems: we modify Spotify's Android app to run as a leecher and evaluate the impact of Spotify's distributed peer discovery mechanism on energy consumption (Section 1.2.1) and we design, implement, and deploy a P2P video streaming system where mobile devices are mainly leechers but they are able to upload original content thanks to our novel usage of an existing distributed peer discovery mechanism (Section 1.2.2).

1.2.1 The Impact of Distributed Peer Discovery on Energy Consumption

Spotify [16] is a peer-assisted music streaming service, offering over 20 million tracks to over 24 million users in 28 countries.

Currently, Spotify uses its P2P network as the main source of the music played by its desktop client (80% of the total traffic), while its own servers primarily handle cases where the P2P network is unable to provide data in a timely fashion to a client.

On its *mobile platforms* (Android, iOS, Windows Phone, etc.) this is not the case, however. These clients, get 100% of the data directly from Spotify's servers. This situation is not unique to Spotify —we are not aware of any large-scale P2P system where mobile platforms are well integrated— despite the dramatic increase in use of such devices.

On the one hand, enabling P2P on these devices would greatly offload Spotify's servers, reducing distribution costs. On the other hand, mobile app developers must make sure their app does not drain the device's battery and not overuse mobile data traffic, which is usually capped.

In this thesis, we implement and evaluate a leecher version of the Spotify client on Android. As explained above, a leecher does not upload data to other peers, thus the amount of music-related data transferred is expected to be similar to the standard client-server Spotify app. The difference is not in the amount of bytes, but its source: peers (plus a fallback Spotify server) in one case, a Spotify server in the other.

A naive developer may disregard the impact of peer discovery on energy consumption, given that the amount of traffic generated by any peer discovery mechanism is negligible compared to the typically large size of multimedia data transfers. On mobile networks, however, traffic patterns can have a greater impact on energy consumption than the number of bytes or number of packets transferred.

Spotify's distributed peer discovery mechanism is based on a gossip-like un-

structured overlay, similar to Gnutella's. Given its low overhead [16], the traffic it generates —ping and search messages— is not expected to be an important factor when considering the total amount of traffic, when compared to multi-megabyte music tracks. It has, however, a tremendous impact on energy consumption, and thus battery drain.

In Chapter 10, our experimental results show that Spotify's distributed peer discovery's traffic pattern —continuous flow of small messages— keeps the device's 3G radio on a high-power state, dramatically increasing energy consumption (about 85% increase).

Then, we implement a simple backwards-compatible modification on the client's peer discovery mechanism which brings down energy consumption close to the standard non-P2P app while downloading most of the music data from peers.

On Wifi, the energy increase when P2P is enabled is much smaller (around 10% increase) and it is slightly reduced when our modification is applied.

Our peer discovery modifications bring energy consumption close to the standard non-P2P app both on 3G and Wifi and they can be deployed immediately because they are backwards-compatible. These modifications have, however, side effects such as increasing overall churn in the overlay.

That is why we also propose backwards-incompatible modifications to the protocol to (1) further reduce energy consumption and (2) eliminate side effects on the overlay, such as spurious churn.

1.2.2 P2P Video Streaming from and to Mobile Devices

The concept of *mobile leechers* fits well a system like Spotify, where content is controlled centrally and clients consume content but are not allowed to share their own user-generated content with others.

Open P2P-based file-sharing systems, however, are not centrally controlled by any one entity and any member of the P2P network can publish content and offer it to others at any moment.

In this thesis, we design and implement a distributed peer discovery mechanism that enables content injection from mobile devices that (1) bypass common connectivity artifacts on Wifi and mobile networks and (2) keep the device in *leecher mode* when possible to reduce energy consumption and traffic volume.

As in the previous system, we rely on desktop peers to serve mobile leechers. In this open protocol system, based on the proposed IETF standard PPSP [20], we rely on *boosters* running on desktop computers.

A *booster* (a *seeder* in BitTorrent's terminology) is a peer that is configured to find and distribute content. We define the term booster broadly to allow for a broad spectrum of boosters that may be combined with incentive mechanisms, both centralized (e.g., private BitTorrent trackers [21–23]) and distributed (e.g., BarterCast [24, 25] and Dispersy [26]). In this thesis, however, we will mainly focus on *altruistic boosters*.

Altruistic boosters are configured to help distribute content published by a

given set of sources/publishers. That is, a boosting volunteer offers its resources to distribute content to other peers, some of them mobile leechers.

We now give an example to help understand the system and why a user may volunteer to run a booster. A person (the publisher) publishes videos where she shares her (possibly controversial) political opinions. Someone who agrees with her (or simply want to counter censorship against her[7]) can then configure a booster to replicate and redistribute her videos. That way, a large audience of mobile users can stream the video, as long as there are enough peers and boosters supporting its distribution.

In Chapter 11, we present our proposal and provide technical details about our prototype: Tribler Mobile.

1.3 Thesis Contribution

- Improvement of lookup performance on an existing large-scale DHT overlay. Lookup performance achieved is one order of magnitude better compared to previous attempts in the literature and closes the gap between the performance observed in large-scale overlays on the open Internet and small-scale overlay deployments and simulations.

- 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.

- We 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 study the impact of peer discovery mechanisms on mobile devices' energy consumption, observing that different P2P parameters and wireless technologies have a great impact on energy consumption. We show backwards-compatible modifications that significantly reduce energy consumption on 3G and propose backwards-incompatible modifications to better integrate mobile devices into P2P networks and further reduce energy consumption.

- We are the first to propose, implement, and deploy a complete system where mobile devices can publish content directly into the P2P network and let other peers distribute the content to a potentially large audience (including other mobile devices). In the process of building this system, we develop a reverse peer discovery mechanism that addresses two major issues: connectivity and energy.

[7] "I may not agree with you, but I will defend to the death your right to make an ass of yourself." —Oscar Wilde

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 P2P systems.

1.4 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. Finally, Chapter 5 concludes.

Chapter 2

Background

2.1 P2P-Based Scalable Services

Scaling services used to be as simple as replacing one server with a more powerful one. At first, improvements in processing speed and increases in memory and storage capacity were enough. Then, servers started to incorporate multiple processors, and even multiple multi-core chips —this tendency is so pervasive that many mobile phones have multi-core processors nowadays.

At some point, a single powerful server was not powerful 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 distributed systems. Finally, we currently witness the dramatic raise of *cloud computing*.

Cloud computing providers deploy inter-connected massive data centers, each one consisting of thousands of inexpensive general-purpose machines. These large-scale distributed systems are possible thanks to distributed algorithms that coordinate all these machines. Examples of distributed systems that scale to data-center levels are: Amazon's Dynamo [27], Google's MapReduce [28], Microsoft's Dryad [29], and Apache's Hadoop [30] and Cassandra [31].

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 [32], 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 [16] (music streaming service). In these systems, users' machines collaborate directly with each other using *peer-to-peer* protocols.

[1] `http://www.skype.com/` (Nov. 2013)
While Skype started running distributed algorithms on the end-users' machines, these tasks are now mainly performed on the cloud (i.e., distributed algorithms running on data-centers).

2.1.1 Peer-to-Peer Systems

Peer-to-peer (P2P) systems, the subject of this thesis, are composed of 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 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 those 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 a 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.

2.1.2 Distributed Peer Discovery

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 these are the communities surrounding 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 [33], BitTorrent tracker [34], eDonkey/eMule's *servers* [35]— in an otherwise distributed system.

There were also early attempts to build large-scale fully-distributed systems based on unstructured overlays (Freenet [36], Gnutella [33, 37]) but performance [38] and scalability [39] issues made them uncompetitive compared to BitTorrent and eMule.

Distributed hash tables (DHTs) 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.

Furthermore, BitTorrent has a third peer discover mechanism, called peer exchange (PEX). PEX resembles an unstructured overlay without routing capabilities.

Coming back to the commercial P2P systems, Spotify [16] uses two independent peer discovery mechanisms: a centralized tracker similar to BitTorrent's, and an unstructured overlay similar to Gnutella's with routing limited to two hops for lookups.

While these peer discovery mechanisms are still in use and have served well, we need to address major issues to adapt them to new requirements such as video streaming (Section 1.1) and the integration of smartphones (Section 1.2) in P2P systems. This thesis contributes towards this goal.

2.2 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 an 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 [40]. 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 changes in system size and network performance, a critical property since nodes join and leave the DHT overlay independently of each other and the underlying network links between nodes may

fail or suffer performance degradation.

These properties have attracted much attention from researchers, who have proposed different DHT designs. Well-known DHT designs include: CAN [41], Chord [42], Pastry [43], Tapestry [15], and Kademlia [44]. An extensive survey by Urdaneta et al. [45] covers all these DHT overlays from a security point of view.

All these DHT designs have been formally analyzed and tested in controlled environments such as simulators and small-scale deployments as well as in data-center applications such as Dynamo [27].

Deployment of DHT-based applications have received much less attention, though. Two remarkable exceptions are OpenDHT [11] and CoralCDN [12]. 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[3], providing a great opportunity for researchers to identify and analyze issues related to real-world deployment. On the other hand, both of them were 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 [10] in both cases [12, 46]).

2.2.1 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 peer discovery for P2P streaming 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 [44]) 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 [47], 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 [14, 48–50]. ADHT [6, 7] and MDHT [6] 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. [5] estimate its size is between six and eleven million nodes[4].

[2] OpenDHT was discontinued in 2009.
`http://opendht.org` (Nov. 2013).

[3] CoralCDN is still running, although active development has stopped as of August 2012.
`http://www.coralcdn.org/overview/` (Nov. 2013)

[4] A real-time estimation is available at
`http://dsn.tm.uni-karlsruhe.de/english/2936.php` (Nov. 2013).

By focusing on a single overlay network, 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 a viable option for building large-scale distributed systems on the Internet. That is, we need to evaluate a deployed DHT overlay which, considering the above-mentioned 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.

The only measurement of lookup performance on MDHT we are aware of yielded a median lookup time of around one minute [6]. The best lookup performance ever reported on these large-scale DHT overlays is a median of 1.5 seconds, achieved by Steiner et al. [49] on KAD.

Applications like video streaming 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"*. Since peer discovery must be performed before any peer-to-peer data transfer, peer discovery is in the critical path, thus high lookup latencies will negatively impact user experience.

2.2.2 Related Work on Improving DHT Performance

Now we take a historical look at researchers' attempts to reduce lookup latency in DHT networks.

Li et al. [51] 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. [52] 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. [46] showed that an overlay deployed on 300 PlanetLab hosts can achieve low lookup latencies (median under 200 ms and 99^{th} percentile under 400 ms). Dabek et al. [53] achieved median lookup latencies between 100–300 ms on an overlay with 180 test-bed hosts.

Crosby and Wallach [6] 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. [7] reduced ADHT's median lookup latency from 127 to 13 seconds by increasing

the lookup cost three-fold.

Stutzbach and Rejaie [48] 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. [49] 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.

2.3 P2P Streaming on Mobile Devices

Internet content streaming services are very popular and growing fast. Not only on large-screen wire-connected desktops but also battery-powered wireless-connected mobile devices. In fact, all trends indicate that mobile devices already account for a large part of streaming, and it is growing far faster than on desktop platforms. For instance, mobile devices consumed 41% of YouTube's total traffic in the third quarter of 2013, up from 25% in 2012 and just 6% in 2011.[5]

Streaming content to a large audience is far from trivial. In general, it is impractical for an individual or a small company to deploy its own content distribution system. While there are platforms that will distribute your content for a fee (e.g., content distribution networks) or placing ads (e.g., YouTube), there is an alternative: *P2P content distribution*.

On the desktop environment, where desktop computers and laptops are commonly connected to a power source and connected to the Internet though Wifi or a wired network, P2P offers the possibility of augment the publisher's distribution capacity with resources from its viewers. That is, the viewers not only consume resources but also contribute their own, increasing scalability and reducing costs to the publisher.

File-sharing communities use P2P to distribute content by each peer contributing its own resources (mainly storage and bandwidth). P2P systems have evolved to be fully-distributed and self-organized, gradually removing centralized services (e.g., tracking and torrent files acquisition in BitTorrent, which initially relied on centralized services, have now been fully decentralized).

Yet, mobile devices have not been well integrated on P2P networks. Conventional wisdom suggests that these devices are ill-suited for P2P for several reasons: (1) uploading would drain batteries and deplete 3G monthly traffic allowances too fast for most users, (2) connectivity issues caused by widely-deployed network address translator (NAT) gateways, and (3) for some publishers, client-server (e.g., content distribution networks (CDNs)) distribution costs are low enough to disregard investments in P2P-based optimizations.

[5] `http://www.computerworld.com/s/article/9243331/` (Nov. 2013)

Spotify, a commercial music streaming service, is a good example to illustrate this point. On desktop platforms, Spotify uses P2P mechanisms to increase scalability and reduce costs. According to their own measurements, 80% of the music is delivered by its P2P network [54]. On mobile platforms, however, all music is currently delivered by Spotify's backend services. These backend services [55] run at Spotify's data centers.

The following sections introduce background regarding network-related energy consumption on mobile devices (Section 2.3.1), related work on mobile P2P applications (Section 2.3.2), and NAT gateways on wireless networks (Section 2.3.3).

2.3.1 Network-Related Energy Consumption on Mobile Devices

One of the main concerns for users and smartphone app developers is battery life. The proliferation of battery-hungry apps make smartphones' batteries last shorter and the availability of energy monitors give users the tools to identify and uninstall apps that provide low value per joule.

Several studies have investigated energy consumption on mobile devices. Zhang et al. [56] and Yoon et al. [57] studied in detail power consumption caused by the different hardware components of a smartphone, including Wifi and 3G.

Carat [58] is a mobile app that collect energy-related information from thousands of users. Carat users are presented with an estimation of how much longer battery will last if a specific app is killed. Users can kill apps directly from Carat's interface.

In this thesis, we focus on power consumption caused by network activity in Wifi (IEEE 802.11g) and 3G. While a brief description of their power properties is given in this section, we refer to the studies above for further details.

2.3.1.1 3G

There are three power states in the 3G radio resource controller (RRC): IDLE, FACH, and DCH. To transmit data, the radio needs to be powered up to DCH where the device acquires a dedicated channel for communication. After a few seconds of inactivity, the radio is demoted to FACH, where it is assigned a shared channel. In FACH, power consumption is about half of DCH and the state can be promoted to DCH quickly if more data needs to be transmitted. After a few further seconds of inactivity, the radio is powered down to IDLE where it consumes very little power.

Figure 2.1 illustrates 3G power states on an Android smartphone. Approximately two seconds into the graph, the phone sends a TCP packet to a server (red dots represent packets captured by tcpdump). To transmit the packet, the 3G radio transitions into DCH, increasing power consumption. Around two seconds, the packet reaches the server, which sends a TCP ACK back. Finally, about four seconds into the graph, the smartphone receives the TCP ACK.

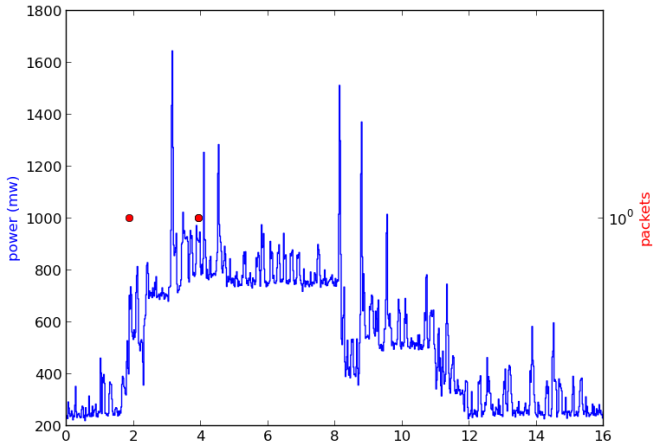

Figure 2.1: A ping-response exchange (packets are represented as red dots at 2 and 4 seconds) shows power transitions on 3G: DCH (2–8 seconds) and FACH (8–12 seconds). Notice the latency of around 2 seconds. This figure also appears in Chapter 10.

This example illustrates the long initial delay (actual RTT was below 100 ms) applications are subjected to. Precisely, to counter such long delay for subsequent packets, the interface stays in DCH for around four seconds and an extra four seconds in FACH before it powers down back to IDLE. Inactivity timeouts are controlled by network operators and can vary widely.

2.3.1.2 Wifi

Wifi data transfer is much more efficient than 3G, specially when power saving mechanism (PSM) is enabled. PSM lets the radio interface sleep and just wake up at regular intervals to listen for beacons from the access point. These beacons indicate whether the mobile device has received data. This data is buffered by the Wifi access point and can be retrieved by the client when it wakes up.

While PSM adds latency, it saves energy. Modern smartphones support adaptive PSM (PSM-A) which keeps the wireless interface powered for an extra period of time to lower latency and increase throughput. In modern smartphones this extra period (i.e. energy tail) has been estimated to be approximately 200 ms [59].

A detailed description of Wifi power consumption can be found in [60]. A recent survey on energy-efficient streaming further elaborates on power saving methods in Wifi [59].

2.3.2 Mobile P2P & Energy Consumption

There are a number of studies investigating the integration of mobile devices in P2P systems.

Bakos et al. [61, 62] simulated Gnutella on a variety of wireless topologies. Nurminen and Nöyränen [18] measured power consumption of SymTorrent—a BitTorrent client for Symbian—both in 3G and Wifi. They report a higher energy consumption on 3G, which is mainly caused by lower download speed on 3G, thus keeping the radio powered for a longer period of time. They also measure "full peer" versus "client only" but only on Wifi, considering it feasible to run full peers on mobile devices (download and upload) since BitTorrent's tit-for-tat mechanism [19] rewards uploaders with faster downloads, thus decreasing energy consumption through a shorter total downloading time. Notice that tit-for-tat is less relevant in the systems we study in this thesis (Spotify in Chapter 10 and Tribler Mobile in Chapter 11) because they are seeding-rich environments.

Kelényi and Nurminen [63, 64] measure energy consumption of Mainline DHT, BitTorrent's decentralized peer discovery mechanism. They highlight the high energy cost of continuous low-bandwidth traffic on Wifi. They reduce energy consumption by selectively dropping DHT queries, allowing mobile devices to use DHT-based services with minimum energy cost.

A recent study by Nurminen [65] evaluates energy consumption of BitTorrent and BitTorrent's Mainline DHT. While it studies energy consumption using a Nokia device on a 3G network, it focuses on how BitTorrent's incentives affect download time, and thus energy consumption.

Petrocco et al. [66] evaluated the performance of Libswift (reference implementation of PPSP) on Android. The study includes power comparison, using hardware measurements, of Libswift and YouTube playing a video stream, using hardware measurements. They did not consider peer discovery, focusing on streaming on Wifi.

RapidStream [67] is P2P video streaming prototype on Android. The paper focuses on design and practicalities with the Android platform. Superficial energy measurements (phone's battery indicator) were given in the paper.

There is a considerable number of design papers and network performance and mobility issues on mobile P2P but do not address energy consumption. For instance, Eittenberger et al. [68] evaluate BitTorrent on WiMax and Berl and Meer [69] consider incentives when integrating mobile devices as leechers in the eDonkey network.

A survey by Hoque et al. [59] showed numerous approaches at optimizing energy consumption of multimedia streaming on Wifi, 3G and LTE networks found in the literature, categorized by layer: physical, link, application, and cross layer. Hoque and his colleagues went on to propose their own approach using crowd-sourced viewing statistics and evaluated it on 3G and LTE networks [70]. On these networks, there is a trade-off between (1) pre-fetching large chunks at the risk of downloading useless data if the user stops watching, (2) and fetching small chunks, at the risk

of increasing total energy consumption because wireless interfaces stay powered for several seconds after each data transfer. Although these studies focused on client-server streaming, they provide insights that are also applicable to P2P streaming on mobile devices.

2.3.3 NAT Gateways

Network address translator (NAT) gateways are used to allow multiple devices to access the Internet using a single IP address. Practically, every Wifi-enabled router (including ADSL, and cable modems provided by Internet service providers) features NAT functionality with few exceptions (e.g., KTH's Wifi network provides public IP addresses). That is, most Wifi connections go through a NAT gateway.

Hätönen et al [71] studied 34 different home gateway models and observed noticeable differences among NAT implementations, reporting that no gateway used the parameter values recommended by the IETF standard for NAT gateways [72].

Mobile networks also use NAT mechanisms, mainly to multiplex an ever-scarcer number of IP addresses. Mäkinene and Nurminen [73] characterized the NAT policies of six major mobile operators, observing that existing TCP NAT traversal techniques work in the majority of these networks. Wang et al. [74] developed a mobile app which allowed them to collect data regarding NAT and firewall policies deployed in over 100 mobile ISPs. Less than a quarter of the these mobile ISPs provided public IP addresses.

NAT gateways supports relatively well client-server applications where clients initiate connections and the server has a public IP address. When a connection is established by a host behind a NAT gateway (client), the gateway will forward packets between client and server, for the most part.

On P2P networks, peers should ideally be able to establish connections to each other because each peer is able to both consume and offer services. On the Internet, however, NAT gateways severely restrict connectivity, making it hard to establish a connection to a peer behind a NAT gateway.

In general, NAT gateways hinder P2P performance [75]. When the only peer with a copy of the content is behind a NAT gateway, content distribution might never start. We encounter this problem when we stream content *from* a mobile device, as we do in Chapter 11.

There are many NAT traversal mechanisms to allow hosts behind NAT gateways to establish direct connections to each other. For instance, NatCraker [76], Usurp [77]. Halkes and Pouwelse's UDP NAT puncturing [78] provide an overview of UDP NAT traversal techniques used on the Internet. While Spotify and our Tribler Mobile do not does not use any of these mechanisms, we are considering integrating some of these mechanisms in future versions of Tribler Mobile.

Chapter 3

Problem Definition

Although the problems presented in this thesis are framed in a context where the practical goal is to build or/and improve particular large-scale P2P systems, the problem definitions we now present are not restricted to these specific practical goals. Instead, these problems are generic problems of adapting an overlay to: (1) the underlying infrastructure (open Internet, NAT gateways, mobile devices) and (2) the requirements defined by the application using the overlay network (P2P streaming in our case).

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

- **Underlying Connectivity**

  It is well known that connectivity on the Internet is far from ideal. Nevertheless, many distributed systems have been designed on the implicit 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 open Internet helps to understand the size of the problem, the potential effects of the underlying connectivity artifacts on the P2P overlay, and to devise mechanisms to adapt the overlay to the underlying network environment.

- **DHT 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 overlays, but truly sub-second results where only a minimal fraction of the lookups (e.g., less than one per cent) take over one second.

- **Scalability and Locality**

  By using randomly distributed keys and nodeIDs from the identifier space, all DHT nodes are expected to be responsible for a similar amount of keys. 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 sizes 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 a locality-aware mechanism which return results close to the requester's location. These mechanisms would decrease inter-ISP traffic, potentially benefiting users and ISPs.

- **P2P Streaming on Mobile Devices**

  Mobile devices' resources (mainly battery and bandwidth) are commonly limited and expensive. Thus, many conclude that these devices are ill-suited to participate in P2P networks. While challenging, we believe that mobile devices can be well integrated in P2P networks. To that end, we need to analyze mobile device's resources in the context of P2P systems and design modifications that adapt P2P systems to the new underlying infrastructure.

- **P2P Streaming *from* a Mobile Device**

  P2P streaming *from* a mobile device is particularly challenging, not only due to the cost in battery and uplink traffic, but also widely-deployed NAT gateways in Wifi and mobile networks.

  Ideally, a battery-powered uplink-limited device would transfer a single copy of the content to other (less limited and more efficient) peers and let them redistribute the content at much lower global cost.

- **Backwards Compatibility**

  Deploying a large-scale system on the Internet is hard, but once it has reached critical mass it becomes *the* standard. It then becomes a great hurdle for any new backwards-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 backwards compatible modifications to take advantage of existing large-scale P2P deployments. The advantages are clear and include: (1) the possibility of studying and testing our modifications on real-world large-scale systems and (2) the potential impact of our research on production systems. This approach also introduces challenges: (1) backwards-compatible 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 host 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 our modifications. Trade-offs need to be clearly presented so users and developers can determine which configurations better fulfill their requirements.

  Evaluation tools are necessary to measure the impact of different modifications and explore these trade-offs. We release our evaluation tools as open-source software.

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.
- F. Osmani, V. Grishchenko, R. Jimenez, and B. Knutsson
  "Swift: the missing link between peer-to-peer and information-centric networks"
  First Workshop on P2P and Dependability, ACM
  Sibius, Romania, May 2012
- R. Jimenez, G. Kreitz, B. Knutsson, M. Isaksson, and S. Haridi
  "Integrating Smartphones in Spotify's Peer-Assisted Music Streaming Service"
  Submitted for publication.
- R. Jimenez, A. Bakker, B. Knutsson, J. Pouwelse, and S. Haridi
  "Tribler Mobile: P2P Video Streaming from and to Mobile Devices"
  Submitted for publication.

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 [2] and appears in Chapter 6.

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 [15] which addresses both scalability issues caused by popular keys and provides locality-aware features, with the intention to reduce inter-ISP BitTorrent traffic.

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 [3] and appears in Chapter 7.

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 perform 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, we propose mechanisms 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 has proven to substantially improve lookup performance, as shown in next section.

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

The work on achieving sub-second lookups on a multimillion-node DHT overlay has been published in a conference paper [79] and appears in Chapter 8.

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, 99^{th} percentile below 600 ms). The modifications needed to achieve such performance are completely backwards-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 P2P-Based Implementation of ICN

The work on a P2P-based implementation of information-centric networks (ICN) has been published in a workshop paper [80] and appears in Chapter 9.

We propose our existing Swift protocol, which would be later called PPSP [20] in the IETF standarization process, as an approach to deploy a ICN system incrementally. While Swift inherits many of its properties from BitTorrent, it introduces several improvements. Unlike BitTorrent, Swift is designed for multimedia streaming, both on-demand and live; although it can also be used for bulk P2P data transfer.

Since Swift is based on UDP, it is easier to traverse NAT gateways and achieve lower delays. Swift uses a hashing scheme (Merkle hashes) that greatly improves the dissemination of integrity metadata, greatly reducing user-perceived startup delay compared to BitTorrent, special when considering large amounts of data such as a high-definition feature film. While the few hashes needed by a Swift peer to start checking integrity are likely to fit in a single UDP, a BitTorrent peer must

obtain every single hash (several kilobytes packed in a *.torrent file*) before being able to perform any integrity check.

Moreover, Merkle hashes provide naming properties very similar to those proposed in other ICN systems, making Swift a good candidate for piece-meal adoption on the current IP-based network infrastructure, unlike other ICN proposals that require a clean-slate redesign of the networking infrastructure.

While Swift is designed to support different kinds of peer discovery mechanisms, we present a complete solution using BitTorrent's Mainline DHT.

Further technical details can be found in our technical report [81] (not included in this thesis).

4.6 Integrating Battery-Powered Devices into a Peer-Assisted Streaming Service

The work described in this section has been submitted for publication and appears in Chapter 10.

In this work, we explore the feasibility of integrating mobile devices into a large P2P network, considering the mobile devices' limitations. We are mostly interested in *mobile leeching*. That is, mobile devices join the P2P network and download from other peers (running on desktop Spotify clients) while they do not upload data to others.

We use Spotify as study case. Spotify is a peer-assisted music streaming service. Currently, desktop Spotify clients participate in a P2P network which delivers around 80% of the music data. Mobile clients, however, do not currently participate in the P2P network and retrieve all content from Spotify's servers.

We enable P2P on Spotify's Android app in a way that it participates in the P2P network as desktop clients, then match local traffic to power consumption. Our empirical measurements indicate that distributed peer discovery has a great impact on energy consumption on 3G networks, while Wifi networks are more power-efficient at handling continuous low-bandwidth traffic associated to distributed peer discovery.

We implement backwards-compatible modifications on the Android app to reduce the amount of energy related to peer discovery. Then, we deploy the modified client on the actual P2P network and observe that we achieve large energy reductions on 3G networks.

Finally, we propose backwards-incompatible modifications to further reduce energy consumption and limit spurious churn generated as side-effect of our modifications.

4.7 P2P Content Streaming *from* Mobile Devices

The work described in this section has been submitted for publication and appears in Chapter 11.

While the previous section addresses solely P2P streaming *to* mobile devices, the work presented in this section also addresses P2P streaming *from* mobile devices.

We are the first to propose, implement, and deploy a complete system where mobile devices can publish content directly into the P2P network and let other peers (running on servers and desktop computers) distribute the content to a potentially large audience (including other mobile devices).

Our contributions include a reverse peer discovery mechanism that addresses two important issues. First, it allows mobile devices to traverse NAT gateways — these gateways are widely-deployed on Wifi and mobile networks. Second, it saves battery because the mobile device is never registered on the peer discovery service, thus other peers will never contact the mobile device —the device's radio must switch to high-power state when receiving packets.

Our prototype has been built incrementally. Whenever possible we have used existing proven-to-work components, implementing backwards-compatible modifications to adapt each component to our requirements. For instance, our reverse peer discovery mechanism is backwards-compatible with BitTorrent's Mainline DHT, allowing us to leverage this multi-million DHT overlay.

4.8 Source Code

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

4.9 Individual Contribution

I am the main author of all the papers included this thesis and I led the work from the initial idea to writing. I also wrote most of the code required to build/modify the systems under study and to run the experiments.

The only exception is paper D, where Victor Grishchenko provided most of the Swift implementation and Flutra Osmani led the work required to write the paper. In paper D, I contributed to the general discussions and paper edition throughout the process. In particular, I contributed to frame the topic in introduction and I wrote the text related to DHT-based routing.

Instead of listing my individual contributions, I list the contributions of others for all papers except D:

- 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.

- In paper E, Gunnar Kreitz and Marcus Isaksson provided background on Spotify's architecture (P2P protocols in particular) and guided my research throughout my stay at Spotify. Co-authors provided valuable feedback on paper drafts and assisted in its edition.
- In paper F, Johan Pouwelse and Arno Bakker participated in the design discussions. Bakker made all necessary modifications to `libswift` and built the initial Android GUI design, which I modified to integrate Twitter and share videos from the smartphone. Co-authors provided valuable feedback on paper drafts.

Chapter 5

Conclusion

This thesis has presented our contributions in the understanding of large-scale P2P-based systems on the open Internet, where connectivity is far from ideal and the amount of mobile devices is already large and it is growing at a fast pace.

We have studied important DHT 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's 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, have proven to dramatically improve lookup performance.

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 feasible to integrate these modifications into a Kademlia-based overlay such as Mainline DHT.

Connectivity artifacts are also common on mobile devices because NAT gateways are widely-deployed on Wifi and mobile networks. We are the first to propose, implement, and deploy a P2P streaming system that allows mobile devices to publish directly into the P2P network, even when they are behind restrictive NAT gateways. Our backwards-compatible modifications in an existing peer discovery system provide this functionality while efficiently using battery and data traffic, two of the main current limitations on mobile devices.

Given the importance of energy consumption on mobile devices, we studied energy consumption related to Spotify's distributed peer discovery mechanism. We

observe that enabling P2P on Spotify's Android app consumes a great amount of energy on 3G, while Wifi handles peer discovery traffic much more efficiently. Our backwards-compatible modifications dramatically reduce energy consumption while creating spurious churn. We also propose backwards-incompatible modifications that would further reduce energy consumption without creating spurious churn as a side-effect.

Conscious of the importance of powerful and reliable tools on the long-term study of large-scale P2P systems, 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/thesis/` and
`http://rauljimenez.net/kth/thesis/`.

Bibliography

[1] R. Jimenez, "Kademlia on the open internet : How to achieve sub-second lookups in a multimillion-node dht overlay," 2011. Licentiate Thesis in Communication Systems, KTH.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] S. A. Crosby and D. S. Wallach, "An analysis of bittorrent's two kademlia-based dhts," tech. rep., Technical Report TR07-04, Rice University, 2007.

[7] 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.

[8] L. Liquori, C. Tedeschi, L. Vanni, F. Bongiovanni, V. Ciancaglini, and B. Marinković, "Synapse: A scalable protocol for interconnecting heterogeneous overlay networks," in *NETWORKING 2010* (M. Crovella, L. Feeney, D. Rubenstein, and S. Raghavan, eds.), vol. 6091 of *Lecture Notes in Computer Science*, pp. 67–82, Springer Berlin Heidelberg, 2010.

[9] 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.

[10] 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.

[11] 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.

[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*, NSDI'10, (Berkeley, CA, USA), pp. 7–7, USENIX Association, 2010.

[13] 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.

[14] 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.

[15] 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.

[16] G. Kreitz and F. Niemela, "Spotify – large scale, low latency, p2p music-on-demand streaming," in *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on*, pp. 1–10, 2010.

[17] M. Goldmann and G. Kreitz, "Measurements on the spotify peer-assisted music-on-demand streaming system," in *Peer-to-Peer Computing (P2P), 2011 IEEE International Conference on*, pp. 206–211, 2011.

[18] J. Nurminen and J. Noyranen, "Energy-consumption in mobile peer-to-peer - quantitative results from file sharing," in *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pp. 729–733, 2008.

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

[20] A. Bakker, R. Petrocco, and V. Grishchenko, "Peer-to-Peer Streaming Peer Protocol (PPSPP)," tech. rep., IETF Internet-Draft, 2013.

[21] J. J.-D. Mol, J. A. Pouwelse, D. H. Epema, and H. J. Sips, "Free-riding, fairness, and firewalls in p2p file-sharing," in *Peer-to-Peer Computing, 2008. P2P'08. Eighth International Conference on*, pp. 301–310, IEEE, 2008.

[22] M. Meulpolder, L. D'Acunto, M. Capotă, M. Wojciechowski, J. A. Pouwelse, D. H. J. Epema, and H. J. Sips, "Public and private bittorrent communities: a measurement study," in *Proceedings of the 9th international conference on Peer-to-peer systems*, IPTPS'10, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2010.

[23] Z. Liu, P. Dhungel, D. Wu, C. Zhang, and K. Ross, "Understanding and improving ratio incentives in private communities," in *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pp. 610–621, 2010.

[24] M. Meulpolder, J. Pouwelse, D. H. J. Epema, and H. Sips, "Bartercast: A practical approach to prevent lazy freeriding in p2p networks," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–8, 2009.

[25] R. Delaviz, N. Andrade, and J. A. Pouwelse, "Improving accuracy and coverage in an internet-deployed reputation mechanism," in *Peer-to-Peer Computing'10*, pp. 1–9, 2010.

[26] N. Zeilemaker and J. Pouwelse, "Open source column: Tribler: P2p search, share and stream," *SIGMultimedia Rec.*, vol. 4, pp. 20–24, Mar. 2012.

[27] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *SOSP*, vol. 7, pp. 205–220, 2007.

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

[29] 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.

[30] T. White, *Hadoop: the definitive guide*. O'Reilly, 2012.

[31] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[32] 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.

[33] 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.

[34] B. Cohen, "BitTorrent Enhancement Proposal 3 (BEP3): The BitTorrent Protocol Specification," 2008.

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

[36] 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.

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

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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`.

[46] 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.

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

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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] G. Kreitz and F. Niemela, "Spotify - Large Scale, Low Latency, P2P Music-on-Demand Streaming," in *P2P'10*, 2010.

[55] R. Yanggratoke, G. Kreitz, M. Goldmann, R. Stadler, and V. Fodor, "On the performance of the spotify backend," *Journal of Network and Systems Management*, pp. 1–28, 2013.

[56] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proceedings of the*

eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, CODES/ISSS '10, (New York, NY, USA), pp. 105–114, ACM, 2010.

[57] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha, "Appscope: Application energy metering framework for android smartphone using kernel activity monitoring," in *USENIX ATC*, 2012.

[58] A. J. Oliner, A. Iyer, E. Lagerspetz, S. Tarkoma, and I. Stoica, "Collaborative energy debugging for mobile devices," *Proc. of USENIX HotDep*, 2012.

[59] M. Hoque, M. Siekkinen, and J. Nurminen, "Energy efficient multimedia streaming to mobile devices — a survey," 2012.

[60] Y. Xiao, P. Savolainen, A. Karppanen, M. Siekkinen, and A. Ylä-Jääski, "Practical power modeling of data transmission over 802.11g for wireless applications," in *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, e-Energy '10, (New York, NY, USA), pp. 75–84, ACM, 2010.

[61] B. Bakos, G. Csúcs, L. Farkas, and J. K. Nurminen, "Peer-to-peer protocol evaluation in topologies resembling wireless networks. an experiment with gnutella query engine,"

[62] B. Bakos, L. Farkas, and J. K. Nurminen, "P2p applications on smart phones using cellular communications," in *Wireless Communications and Networking Conference, 2006. WCNC 2006. IEEE*, vol. 4, pp. 2222–2228, IEEE, 2006.

[63] I. Kelenyi and J. Nurminen, "Optimizing energy consumption of mobile nodes in heterogeneous kademlia-based distributed hash tables," in *Next Generation Mobile Applications, Services and Technologies, 2008. NGMAST '08. The Second International Conference on*, pp. 70–75, 2008.

[64] I. Kelenyi and J. Nurminen, "Energy aspects of peer cooperation measurements with a mobile dht system," in *Communications Workshops, 2008. ICC Workshops '08. IEEE International Conference on*, pp. 164–168, 2008.

[65] J. Nurminen, "Energy efficient distributed computing on mobile devices," in *Distributed Computing and Internet Technology* (C. Hota and P. Srimani, eds.), vol. 7753 of *Lecture Notes in Computer Science*, pp. 27–46, Springer Berlin Heidelberg, 2013.

[66] R. Petrocco, J. Pouwelse, and D. H. J. Epema, "Performance analysis of the libswift p2p streaming protocol," in *Peer-to-Peer Computing (P2P), 2012 IEEE 12th International Conference on*, pp. 103–114, 2012.

[67] P. Eittenberger, M. Herbst, and U. Krieger, "Rapidstream: P2p streaming on android," in *Packet Video Workshop (PV), 2012 19th International*, pp. 125–130, 2012.

[68] P. M. Eittenberger, S. Kim, and U. R. Krieger, "Damming the torrent: Adjusting bittorrent-like peer-to-peer networks to mobile and wireless environments," *Advances in Electronics and Telecommunications*, vol. 2, no. 3, pp. 14–22, 2011.

[69] A. Berl and H. de Meer, "Integrating mobile cellular devices into popular peer-to-peer systems," *Telecommunication Systems*, vol. 48, no. 1-2, pp. 173–184, 2011.

[70] M. A. Hoque, M. Siekkinen, and J. K. Nurminen, "Using crowd-sourced viewing statistics to save energy in wireless video streaming," in *Proceedings of the 19th annual international conference on Mobile computing & networking*, MobiCom '13, pp. 377–388, 2013.

[71] S. Hätönen, A. Nyrhinen, L. Eggert, S. Strowes, P. Sarolahti, and M. Kojo, "An experimental study of home gateway characteristics," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, IMC '10, (New York, NY, USA), pp. 260–266, ACM, 2010.

[72] R. Braden, *RFC 1122 Requirements for Internet Hosts - Communication Layers*. Internet Engineering Task Force, Oct. 1989.

[73] L. Makinen and J. K. Nurminen, "Measurements on the feasibility of tcp nat traversal in cellular networks," in *Next Generation Internet Networks, 2008. NGI 2008*, pp. 261–267, IEEE, 2008.

[74] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang, "An untold story of middleboxes in cellular networks," *SIGCOMM Comput. Commun. Rev.*, vol. 41, pp. 374–385, Aug. 2011.

[75] B. Ford, "Peer-to-peer communication across network address translators," in *In USENIX Annual Technical Conference*, pp. 179–192, 2005.

[76] R. Roverso, S. El-Ansary, and S. Haridi, "Natcracker: Nat combinations matter," in *Computer Communications and Networks, 2009. ICCCN 2009. Proceedings of 18th Internatonal Conference on*, pp. 1–7, 2009.

[77] S. Niazi and J. Dowling, "Usurp: Distributed nat traversal for overlay networks," in *Distributed Applications and Interoperable Systems* (P. Felber and R. Rouvoy, eds.), vol. 6723 of *Lecture Notes in Computer Science*, pp. 29–42, Springer Berlin Heidelberg, 2011.

[78] G. Halkes and J. Pouwelse, "Udp nat and firewall puncturing in the wild," 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. 1–12, Springer Berlin Heidelberg, 2011.

[79] 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.

[80] F. Osmani, V. Grishchenko, R. Jimenez, and B. Knutsson, "Swift: the missing link between peer-to-peer and information-centric networks," in *Proceedings of the First Workshop on P2P and Dependability*, P2P-Dep '12, (New York, NY, USA), pp. 4:1–4:6, ACM, 2012.

[81] V. Grishchenko, F. Osmani, R. Jimenez, J. Pouwelse, and H. Sips, "On the design of a practical information-centric transport," tech. rep., PDS Technical Report PDS-2011-006, 2011.

Part II

Research Papers

Paper A

Chapter 6

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

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.

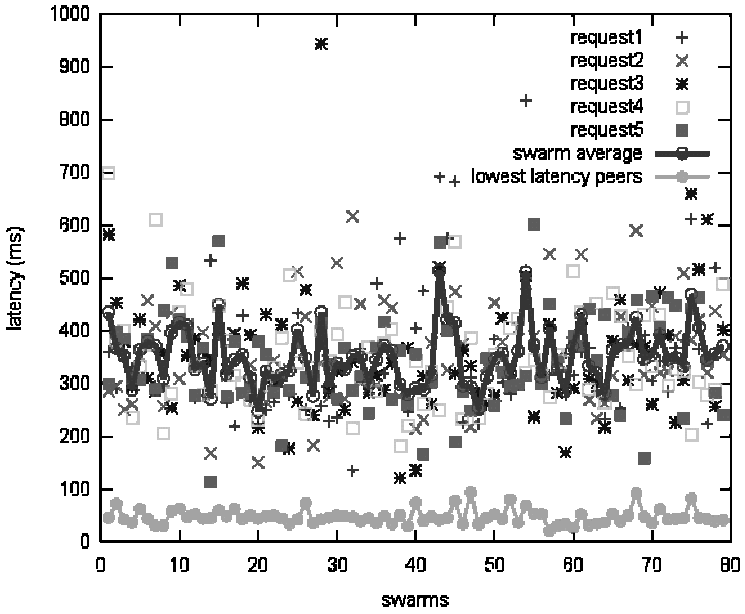

Figure 1: Latency measurements to peers participating in popular BitTorrent swarms

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 7

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

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

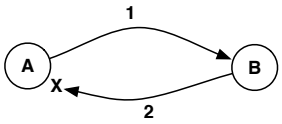

Figure 1. Non-reciprocal connectivity

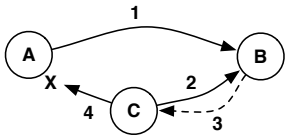

Figure 2. Non-transitive connectivity

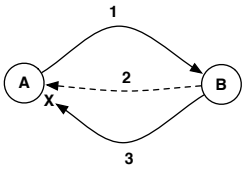

Figure 3. Non-persistent connectivity

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.

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/

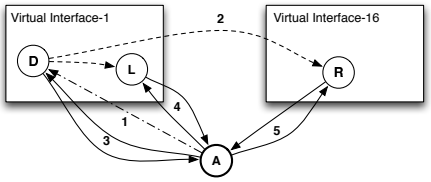

Figure 4. Experiment setup

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.

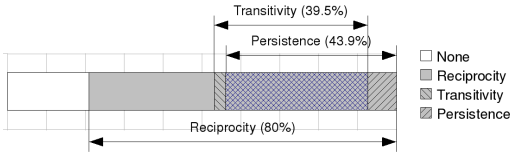

Figure 5. Properties Chart

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.

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.

[10] DDoS stands for Distributed denial of service.

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 8

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

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

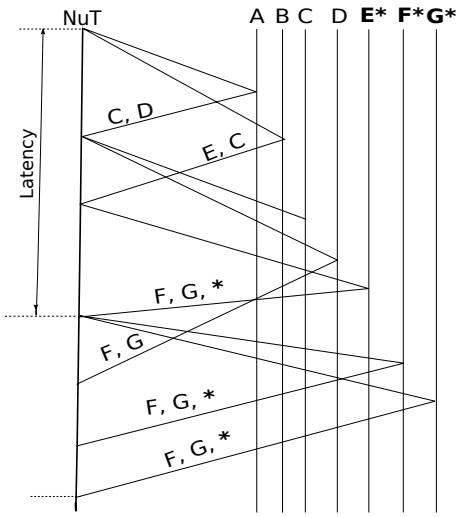

Fig. 1. Lookup performed by node under test (NuT). Letters A–G represent nodes in the overlay. Values are represented with "*".

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/

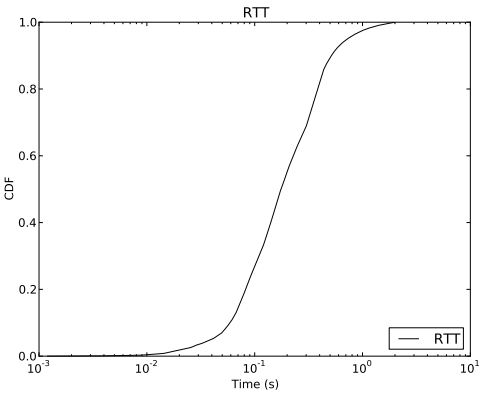

Fig. 2. RTT to nodes in the MDHT overlay (Queries that have not been replied within 2 seconds are considered timed-out, thus excluded from this graph.)

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 (2^{nd} percentile: 2.13 ms, 25^{th} percentile: 94.8 ms, median: 175.2 ms, 75^{th} percentile: 343.6 ms, 98^{th} 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.

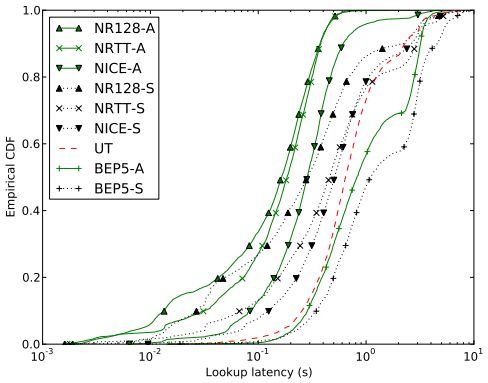

Fig. 3. Lookup latency when retrieving a value from the MDHT overlay

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

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.

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

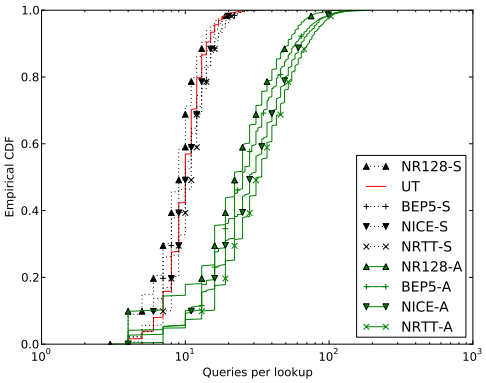

Fig. 4. Lookup cost. (We recognize that distinguishing between individual lines in this graph is hard, but the difference between standard (*-S and UT) and aggressive (*-A) lookup is clear. Also notice the tendency of NR128-* and NRTT-* to have lower and higher cost than the rest, respectively.)

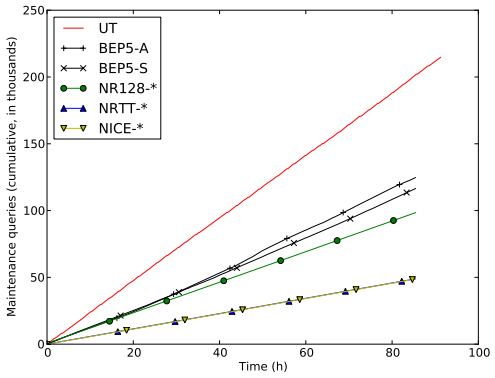

Fig. 5. Cumulative maintenance traffic during the entire experiment

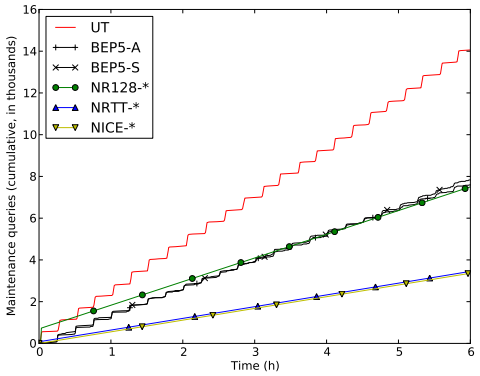

Fig. 6. Cumulative maintenance traffic during the first 6 hours

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

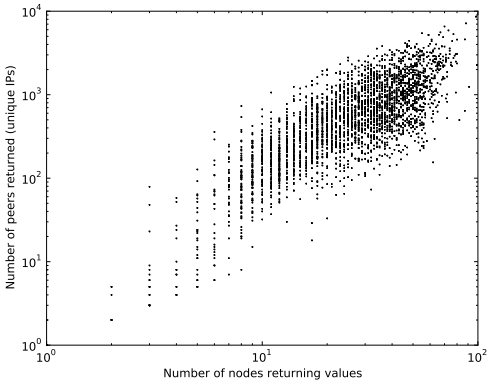

Fig. 7. Peers versus nodes returning values

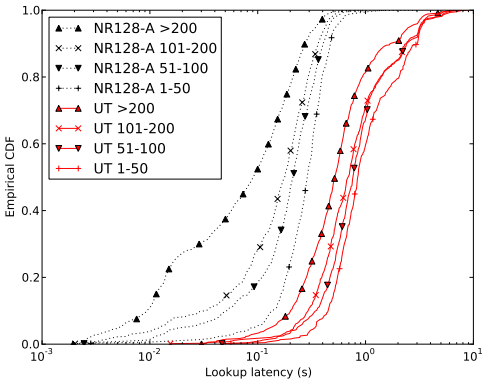

Fig. 8. Lookup latency versus swarm size for UTorrent and NR128-A. Both implementations perform better on larger swarms.

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

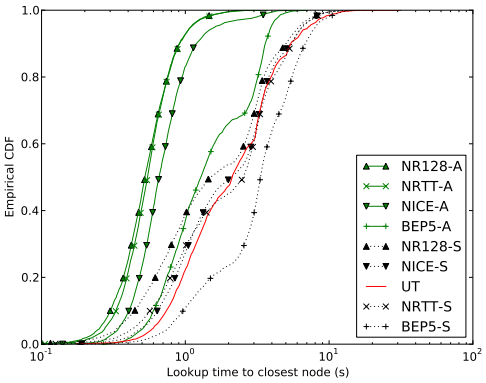

Fig. 9. Lookup latency to reach the closest node to the key

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 99$^{\text{th}}$ 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.

Paper D

Chapter 9

Swift: The Missing Link Between Peer-to-Peer and Information-Centric Networks

F. Osmani, V. Grishchenko, R. Jimenez, B. Knutsson

In *Proceedings of the First Workshop on P2P and Dependability 2012 (P2P-Dep'12)*
May 8–10, 2012, Sibiu, Romania

Swift: The Missing Link Between Peer-to-Peer and Information-Centric Networks

Flutra Osmani
KTH Royal Institute of Technology
Forum 105, SE-164 40
Kista, Sweden
flutrao@kth.se

Victor Grishchenko*
Citrea
Ekaterinburg, Russia
victor.grishchenko@gmail.com

Raul Jimenez, Björn Knutsson
KTH Royal Institute of Technology
Forum 105, SE-164 40
Kista, Sweden
{rauljc, bkn}@kth.se

ABSTRACT

A common pitfall of many proposals on new information-centric architectures for the Internet is the imbalance of up-front costs and immediate benefits. If properly designed and deployed, information-centric architectures can accommodate the current Internet usage which is at odds with the historical design of the Internet infrastructure. To address this concern, we focus on prospects of incremental adoption of this paradigm by introducing a peer-to-peer based transport protocol for content dissemination named *Swift* that exhibits properties required in an Information-Centric Network (ICN), yet can be deployed in the existing Internet infrastructure. Our design integrates components while highly prioritizing modularity and sketches a path for piecemeal adoption which we consider a critical enabler of any progress in the field.

Keywords

Distributed systems, networking, transport protocol

1. INTRODUCTION

In this paper, we introduce *Swift* protocol. *Swift* was originally designed to be a replacement for the BitTorrent protocol and inherits some of the characteristics that have made BitTorrent successful, but was not intentionally designed according to the principles of information-centric networking (ICN) paradigm. It is, thus, until now a product of (unintentional) evolution towards ICN that we now seek to direct and accelerate, while retaining all the properties that make it work well on top of the existing network.

We find the ICN concept to be increasingly reflected in both the way Internet is being used and in how Internet-based services are being implemented today. In many cases, we find that the problems we struggle with in the current incarnation of the Internet are those that ICN design proposals seek to address.

For example, over 90% of today's Internet bandwidth [4] is effectively devoted to disseminating static multimedia content. In order to do so to an increasingly large and geographically diverse audience, various approaches are used. For example, for web-based content, Content Delivery Networks (CDNs) like Akamai use modified DNS servers that generate responses based on the topological/geographical location of the requester. These "tricks" are there to achieve on the current Internet the features that are at the core of ICN.

During the past years, a chain of new network architectures based on the information-centric paradigm (also content-centric, name-oriented, named-data) have been proposed, including CCN [17], DONA [20], NetInf [11], secure naming by Wong et al [23], and content-centric router by Arianfar et al [6] to address the limitations of IP, namely the inability to decouple data from storage, inefficient data dissemination, lack of support for middleboxes, ubiquitous availability of data, and security.

The historical conversation-centric end-to-end model, embodied in the TCP/IP stack, is based on message exchange between pairs of peers, typically *servers* and *clients*. On the other hand, ICN is a paradigm in which focus shifts away from the mechanics of moving bits between peers (end-hosts). Instead, the focus in on the information itself, and the underlying network only a conduit for the information. In a sense, named-data network breaks with the end-to-end abstraction, as there are no *ends* and the entire network is considered a *cloud*, which both stores and serves data.

Similarly, we can see in the evolution of peer-to-peer (P2P) file-sharing technologies how they have adopted ways of managing content that more and more look like ICN. While BitTorrent [10] has always used a SHA-1 hash of the content data to identify that unique content item, it used to be that you also needed a location identifier (the address of the tracker through which the peers hosting the content can be located). However, the current incarnation of BitTorrent instead uses a shared global Distributed Hash Table (DHT) to locate peers using the aforementioned content hash as the key.

The historical Usenet discussion system [5] had all the key information-centric features: logical namespace and unique message identifiers, flood message propagation and caching. The git [3] revision control system represents version history

*Work by Victor Grishchenko was carried out while at the Technical University of Delft

of a project as a directed acyclic graph of revisions, where every revision is identified with SHA-1 hash of its contents; repositories push and pull content, thus forming a network of arbitrary topology.

This tendency towards information-centricity implies a strong demand for a generic named-data substrate that is not reflected yet in the *de jure* network architecture. Given this dissonance, some have proposed a *networking revolution* to dethrone the Internet Protocol (IP) in favor of a clean-slate redesign of the networking infrastructure.

While intellectually attractive, we do not consider such an approach realistic. Not only because it would require the expensive and disruptive wholesale replacement of the existing infrastructure, but more importantly because such a migration is unlikely to happen before the wholesale conversion of applications to information-centric analogues of the current application ecosystems, and that conversion is unlikely to happen until the required infrastructure is in place.

Having made this observation, it seems clear that evolution, not revolution, is the best way towards ICN, and by recognizing and helping this along, we can both make ICN happen sooner and ensure that the ICN approach will be one tested in both lab and real-world settings, and hence "the fittest".

We start with the necessary basic properties of any information centric architecture and determine which of them *Swift* already supports. Further, we determine which information centric primitives *Swift* does not provide and address them by leveraging existing technologies, such as DHTs to find peers and standard IP to route packets.

In this paper, we argue that our modular design addresses the gap between Internet usage and the underlying network, without requiring clean-slate redesigning of the architecture. In particular, *Swift* protocol supports most properties proposed in information-centric architectures like CCN [17], DONA [20], and NetInf [11]. First, *Swift* uses names — flat identifiers — to request content instead of end-point addresses; in addition, it segments named objects in uniquely identified chunks. Second, *Swift* employs per-packet integrity check, enabling any peer in the network to cache and relay content and verify the integrity of each piece. Third, *Swift* avoids transmitting additional metadata and is suitable for live/mutable data, by employing Merkle hashes [21].

Moreover, *Swift* is a receiver-driven chunk-level transport protocol; the receiver may send concurrent requests for chunks to multiple peers in the network in order to enhance its content retrieval rate. To efficiently exploit available bandwidth, *Swift* employs a delay-based congestion control algorithm named LEDBAT [22], and to address the issue of middleboxes *Swift* employs a NAT hole punching mechanism. For peer discovery, *Swift* can use centralized trackers or DHTs; in Section 7 we explain how Mainline DHT (MDHT) can be used to find peers offering the given data object in sub-second time periods.

The paper proceeds as follows. In Section 2 we introduce ICN related work and summarize system description. Section 3 discusses design properties and the resulting separation of transport and internetworking layers. Section 4 describes our variation of the Merkle hashing scheme and its extensions. In Section 5 we introduce a vocabulary of messages that constitutes our protocol. Section 6 describes our UDP-based implementation. Section 7 discusses the implications for peer discovery and packet routing. Section 8 concludes.

2. BACKGROUND

Most proposals on ICN architectures — evolutionary and clean-slate designs — aim to define the main building blocks of an information-centric network. In the CCN [17] design, content names have a hierarchical structure and are constructed according to the standard URI form. Content is requested using an *interest* packet which contains the name of the content. Every content router receiving the *interest* packet checks if the given packet is in its local cache and thus returns a corresponding *data* packet along the reverse path, otherwise, it forwards the *interest* to the correct interface using longest prefix matching. A similar approach for content retrieval is reflected in the PSIRP architecture [2], albeit it uses flat instead of hierarchical names to address content.

Some recent ICN projects adopt flat, self-certifying, labels to name content. Initially employed in DONA's design [20], flat names are used by the route-by-name protocol (devised on top of the IP layer) to request content. Similarly, efforts by Dannewitz et al [11] and Wong et al [23] explore secure naming schemes to ensure the data is persistent and not accessed by unauthorized users.

Self-certifying (flat) names have been criticized for their lack of scalability (cannot be aggregated), flexibility, and lack of security during the translation of flat names to human-readable names. However, recent work by Ghodsi et al [13] argues that self-certifying names exhibit better security properties than human-readable names because 1) they can handle better denial-of-service attacks — the network knows the binding between the name and the key thus it can verify that a given object is associated with a given name, and 2) may scale better through *explicit aggregation* — using concatenations of the form $A.B.C$, where each letter is a name itself.

Despite differences in the naming scheme, the necessary mainstay of any name-oriented network architecture is to employ either cryptographic hashes or signatures in order to enable indiscriminate caching of data in the network and the possibility of its retrieval from any available peer. In *Swift*, we employ hashes — Merkle hashes — and argue that they are *sufficient* to perform any transport function. More specifically, Merkle hash trees [21] allow to identify and verify data, thus enabling any peer in the network to request, relay and store data. Furthermore, our variant of hash trees needs no supplementary transfer metadata, rendering transport into a thinner *layer* than usual.

Swift must be implemented by all inter-operating network peers, and its functioning involves cross-layer relaying of the data, known as *internetworking* (see Figure 1). Hence, the required functionality needs to be as simple and formalized as possible. It follows naturally that any rich semantic data names or transfer metadata is unnecessary and should therefore not be part of transport.

We define a natural separation of *Swift* from the upper *naming* part which deals with problems inherently semantic and the lower *internetworking* layer. Leveraging existing deployed routing infrastructure and a simple hash-based naming mechanism, *Swift* retrieves pieces of content requested by the receiver from peers in the network — functionality that researchers propose to incorporate in any transport protocol

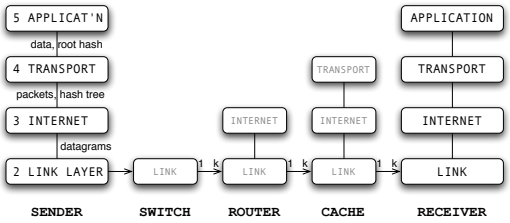

Figure 1: Swift protocol stack: application-dependent queries are translated into requests to the transport layer for particular data — identified with hashes. Transport deals with data streams, their verification and storage.

for information-centric networks [9, 7]. The naming layer is out of scope, thus we make no specific assumptions.

3. DESIGN OVERVIEW

In our design, content is identified by a single cryptographic hash that is the *root hash* in a Merkle hash tree, calculated recursively from the content (see details in the Merkle hash extension document [8]). The ability to verify data against its name allows for storage in the network and retrieval of data from an arbitrary location. Second, as a (packet) network may need to check data integrity piece by piece, possible options boil down to either per-packet signatures, as in CCN, or Merkle hash trees. Differently from signatures, Merkle hash trees provide strict permanent identifiers of static data pieces, so we chose them as the foundation, later extending the approach to dynamic data (see Section 4).

The hashing scheme enables the entire information-centric stack, illustrated in Figure 1, in two ways. First, it allows for a perfect application-to-transport handover. Semantically-rich and application-dependent queries are eventually converted into requests to the transport layer for particular data pieces, precisely identified with hashes. Second, hashing enables information-centric internetworking, i.e. identification and relay of data pieces, data verification, and storage in the network.

As depicted in Figure 1, *Swift* embeds a layer separation scheme very much reminiscent of TCP/IP. Namely, there is a relay internetworking layer that only deals with separate datagrams. On top of it, there is a somewhat more intelligent transport layer that deals with entire data streams, performing verification, caching, and storage.

Any peer running *Swift* may cache content. Technically, there is no difference between a *peer* and a *cache* — they run the same protocol; the conceptual difference lies in the intention: a cache "stores" content to further disseminate it but is not particularly interested in the given content. The caches may be regular peers or peers put in place by ISPs — who are interested in replicating "popular" content within their administrative domains and thus avoid transit traffic and costs to external domains. If operated by ISPs, such caches (interchangeably, peers) may manage the content they offer according to some basic rules, such as LRU or demand.

Discovering peers or caches may be done centrally through trackers or ISP-based trackers, or in a decentralized fashion through PEX or DHTs. We explain how discovering new peers can be done with Peer Exchange in Section 6 or with Mainline DHT in Section 7.

In *Swift*, data storage and data verification are highly interdependent and important in terms of security. For example, if a caching peer does not verify data integrity, it makes *cache poisoning* possible. While a final recipient does not accept (drops) incorrect data, an erroneous cache may form a *clot* in the network, preventing the correct data from passing through. Similarly, data verification requires storage in peers to some degree, as Merkle hash trees need accompanying uncle hash chains to be available in order to verify data pieces.

In a sense, hashes replace IP addresses as end-point identifiers. A receiver uses a root hash to "open" the connection to the network and retrieve the data. The receiver requests specific pieces of data using a novel method called *bin numbers* (see details in the RFC document [14]) which allows the addressing of a binary interval of data using a single integer. This numbering mechanism reduces the amount of state that needs to be stored in each peer and minimizes the space required to denote intervals on the wire. Because the receiver directly addresses the data instead of a single end-point at a particular IP location, it has no control over which peer (replica) will respond; the receiver controls the reception of pieces based on local parameters.

4. THE HASHING SCHEME

We modified Merkle hash trees and focused on smooth operation of both vertical (application to transport) and horizontal (internetworking) handovers to ensure that no peer requires third parties to verify bindings between keys and names (as in CCN [17]) or to retrieve additional metadata to perform their function. We ensure their operation is as simple and formalized, as possible. In Sec. 4.3, we extend our basic technique to the cases of live data streams and versioned data.

4.1 64-bit Merkle Trees

We developed a variant of the Merkle hash tree scheme [21] to satisfy three key requirements: (a) per-packet data integrity checks, (b) no additional metadata and (c) suitability for live/mutable data. The general concept is to start with the root hash only, then incrementally acquire data and hashes, while verifying every single step.

First, content is divided into 1KiB chunks named *packets*, except for the tail packet, which may have less than 1KiB of data. A cryptographic hash, such as SHA1, is then calculated on every packet. Second, a hash tree is defined over the complete $[0, 2^{63})$ byte range, which we consider to be a good approximation of infinity in relation to content size. The tree consists of aligned binary intervals called *bins*, i.e. $[i2^k, (i+1)2^k)$. Bins are nested, forming a strict binary tree (see Figure 2). Each tree contains 2^{64} bins of different sizes, including one void and one root bin; the base — the lowest level — of the tree is composed of 2^{10} byte long bins.

The base of the tree (the leaves) accommodates all the data chunks, starting from the left-most leaf. Normally, the base of the tree is wider than the number of chunks, thus the remaining empty leaves in the tree are assigned hash values of zero. In higher levels of the tree (above base), bins contain hashes which are calculated as a SHA1 hash of a concatenation of two — left and right — child (lower-level) hashes. This hashing process iterates until a hash value for

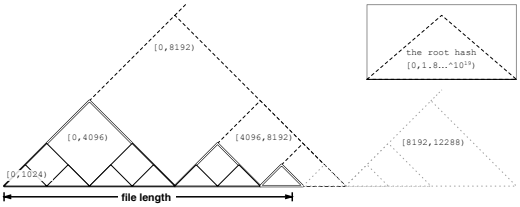

Figure 2: Merkle hash tree constructed for a file of size less than 8KiB. Bins for peak hashes are marked with double lines and they cover the ranges $[0, 4096)$**,** $[4096, 6144)$ **and** $[6144, 7162)$**. Filled bins are marked with solid lines, incomplete bins with dashed lines, and empty bins with dotted lines.**

the root bin is calculated, known as the *root hash*.

Figure 2 illustrates an example where the file size is less than 8KiB long. Its $[8192, 12288)$ empty bin has zero hash by definition, as do the rest of empty bins outside the $[0, 8192)$ range. The root hash covers the entire $[0, 2^{63})$ range; this approach gives us a fixed point of reference when growing the hash tree down from the root.

4.2 Peak Hashes

The concept of peak hashes enables two cornerstone features: file size proving and unified processing of static data and live streams. In addition, they help avoid the usage of additional transmission metadata. Formally, *peak hashes* are hashes defined over filled bins, whose parent hashes are defined over incomplete (not filled) bins. A filled bin is a bin which does not extend past the end of the file, or, more precisely, contains no empty packets.

Practically, we use peaks to cover the data range with a logarithmic number of hashes, so each hash is defined over a "round" aligned 2^k interval. As an example, suppose a file is $l = 7162$ bytes long (see Figure 2). That fits into seven packets ($\frac{7162}{1024} < 7$), the tail packet being 1018 bytes long. For this particular file we will have three peaks, covering $[0, 4096)$, $[4096, 6144)$ and $[6144, 7162)$ ranges (triangles depicted with double lines). The last range might also be written as $[6144, 8192)$ because we round-up to 1KiB packet size.

The number of peak hashes can not exceed $\lceil \log_2 \frac{l}{1024} \rceil$. Practically, peak hashes provide us with more convenient "reference roots", as compared to the root hash which is 53 levels higher than the packets. More importantly, peak hashes allow a sender to quickly *prove* the file size to a recipient who only knows the root hash; otherwise, file size would have to be supplied as a separate metadata piece and thus separately verified, showing up in the protocol and in the interfaces.

4.3 Live Data Streams

In the case of live data streams, the root hash is undefined or, more precisely, transient, as long as new data keeps coming, filling new packets to the right. Hence a transfer has to be identified with a public key instead of a root hash. Keys are more difficult to deal with than hashes, as they have more degrees of freedom. For example, once a key is compromised, any party may rewrite a pre-existing stream. Also, while a hash might be derived directly from the data, a signature can only be verified once known.

Because of such issues, we try to minimize key/signature usage by using the same peak hashes scheme as in the case of static data. Indeed, once a peak hash is defined, it never changes. Thus, we only need a logarithmic number of signatures to sign peak hashes. After that, we may deal with the same Merkle hash tree as before.

Signing the peak hashes only requires the sender to issue the newly formed peak hashes with their signatures attached. On the receiver side, the recipient will only have to check the signature of a new peak hash and whether it matches its child hashes. Such a calculation is incremental and local. Otherwise, if the root hash were to be signed instead, this would require constant re-verification of all the encompassing peak hashes.

Until this point, we assumed that the sender emits data in "round" 1KiB long packets. What if smaller portions of data need to be committed to the network? We do not equal, but we strongly associate our "packets" with link-layer "frames". Thus, once data is worth sending, before it fills a packet, then it also needs a hash and a signature.

5. MESSAGE VOCABULARY

In this section we describe the set of messages that constitute *Swift* protocol. In this section, we refer to it as a *vocabulary* which is instantiated as a transport protocol (as in Figure 1). No particular serialization, encapsulation schemes, or message exchange patterns are specified, beyond the very basic requirements.

A $DATA$ message simply carries pieces of data. A $DATA$ message must carry a bin of data. Specifically, each $DATA$ message contains the bin number of the piece and the piece itself. This way, uniform pieces or multiples of pieces can be processed, making it easier to check the data hash tree at once.

A $HASH$ message carries the necessary hashes that the receiver needs in order to verify the integrity of a piece. We employ the principle of *atomic* datagrams, which means that every piece of data must be verified once received and accepted, otherwise dropped. At this point, the sender must make sure the receiver has every hash needed to verify the incoming data immediately. Finally, it is possible to supply the recipient with parts of the hash tree incrementally, to allow for an amortized and local verification of data.

As we allow for the possibility of data retrieval from multiple peers in parallel, the vocabulary employs $HINT$ and $HAVE$ messages. A $HINT$ (request) message indicates which pieces of data a receiver wants to retrieve, while a $HAVE$ message conveys what pieces of data a sender has available. On incoming data, a receiver uses ACK messages to acknowledge the received pieces; acknowledgements follow the logic of hash trees, which means that data must be acknowledged in bins as well.

We define *channels* as a means to identify ongoing transfers, where each transfer is identified by either a hash or a public key. Channel identifiers are conveyed through the datagram headers.

6. THE UDP IMPLEMENTATION

As previously stated, *Swift* [1] protocol is implemented over UDP; the detailed design is described in an IETF draft [14] and an overview is outlined in a technical report [15]. The

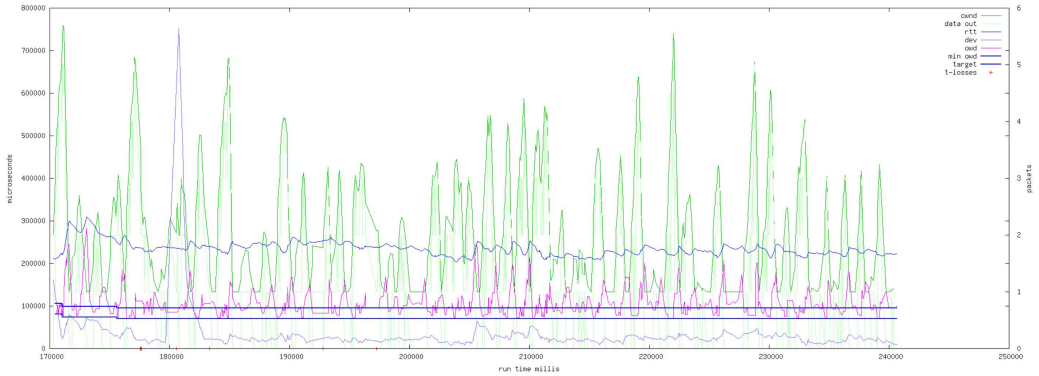

Figure 3: LEDBAT congestion control adjusts the congestion window *cwnd* in order to avoid data losses, by comparing estimated queuing delay against a fixed target delay. The figure depicts experimental results when two nodes exchange pieces of content.

protocol is a direct implementation of the vocabulary (see Section 5), with some additions and extensions.

Messages are serialized as fixed-width fields starting with a single-byte message type field, followed by fixed-width payload fields, such as bin numbers, data, hashes and such. Messages are packed into UDP datagrams. Datagram processing is event-driven, fully implementing the atomic datagram concept. This means that every datagram is either immediately committed to storage or immediately dropped; there are no buffer-re-assembly mechanics.

The UDP implementation employs LEDBAT [22] congestion control algorithm, which allows streams to run virtually lossless under normal conditions. LEDBAT is a delay-based congestion control algorithm which increases/decreases the congestion window based on the estimated queuing delay. It uses an increased queuing delay as indicator of congestion and thus immediately reacts by backing off (decreasing the rate).

Queuing delay in LEDBAT is known as the *one-way delay* (label *owd* in Figure 3) and it is calculated as the difference of timestamped packets between the sender and the receiver. The receiver also maintains a minimum over all one-way delays — *base delay* — which indicates the amount of delay due to queuing. LEDBAT compares this estimated queuing delay against a fixed target delay value (line *target* in Figure 3); the difference determines if the congestion window should be increased or decreased.

Figure 3 depicts how LEDBAT predicts congestion and avoids data losses for a given exchange between two peers (the figure only illustrates a preliminary test performed in a controlled environment with several peers spread across continents). In the testing scenario, one peer acts as a content provider — has the whole content — and other peers (requesters) are interested in the given content. The requesters retrieve the pieces, initially from the only content provider, and later on, they continue retrieving pieces from the participating requesters — who already obtained some pieces of the desired content.

Furthermore, *Swift* implements a unified mechanism of PEX and NAT hole punching functionality [12]. It uses two types of PEX messages — PEX_REQ and PEX_ADD — to retrieve/exchange addresses among the peers, in a gossip fashion. However, PEX messages are transmitted in such a way that they facilitate the communication between peers that are located behind middleboxes: once a peer A introduces peer B to C, it should — within a period of 2 seconds — introduce peer C to B. This mechanism makes *Swift* agnostic to middleboxes.

To guarantee that a receiver can verify every packet, the sender has to prepend it with the missing hashes. In a network with no data loss, the receiver builds the hash tree incrementally, thus every packet of data needs one hash on average. More precisely, every *even* packet needs a hash for its sibling, every fourth also needs a hash for its uncle, every eighth also needs a hash for its parent's uncle, and so forth, thus the average is 1. In practice, some packets are lost, so a prudent sender over-provisions hashes to compensate for possible loss. Thus, the actual traffic overhead of hashes is somewhat above the perfect value of 2% (assuming 20 byte hashes for a 1024 byte packet).

The protocol needs to keep more state on the transfer progress, as data might arrive out of sequence — mostly because data is delivered from different peers in parallel. The *state* must also be communicated over the wire, using unreliable datagrams. We adopted a generic compressed-bitmap data structure named *binmaps* [16], a hybrid of bitmap and a binary tree, which allows to track data at an arbitrary scale, starting from a single packet. Data is requested and acknowledged in bins; this provides the necessary compression and redundancy as continuous data pieces are acknowledged with a logarithmic number of messages.

7. ROUTING AND TRACKING

In order to retrieve data associated to a *root hash*, *Swift* needs to discover peers. This *peer discovery* process is performed by requesting peers from a *tracker*.

Trackers are used in peer-to-peer systems to keep track of peers sharing a given piece of content. The tracker's interface is simple. Peers can request a list of peers for a given content identifier (a *root hash* in *Swift*, an *info hash*

in BitTorrent). Peers also register itself in the tracker to be discovered by others.

Swift can use any tracking mechanism regardless of its particular implementation. Tracker mechanisms used in BitTorrent are prime candidates to be used due to their proven merits on large-scale deployments, but other implementations offering equivalent functionality may be used [24].

BitTorrent's trackers can be centralized or DHT-based. In the first case, the URI of the tracker tracking a given piece of content is necessary. The DHT-based option, on the other hand, forms a global tracking system where all content is tracked, thus no tracker URI is needed.

We favor the DHT-based tracker mechanism due to the scalability of the DHT and the minimization of metadata for *Swift* (no tracker URI is needed, just a *root hash*) to retrieve the data. Scalability is well illustrated by Mainline DHT, the BitTorrent's largest DHT-based tracker on the Internet. Mainline DHT is supported by most of the popular BitTorrent clients, forming a DHT overlay of between 6 and 11 million nodes[19][1]. It is difficult to estimate how many pieces of content Mainline DHT tracks at a given time, but given the size of the BitTorrent ecosystem, even conservative estimations would yield six-digit numbers.

Furthermore, recent measurements [18] have shown that Mainline DHT's response time is consistently low, which makes it suitable for latency-sensitive applications such as on-demand video streaming.

8. CONCLUSION

In this paper, we presented a peer-to-peer based transport protocol for content dissemination named *Swift* and argued that the protocol exhibits ICN properties that help close the gap between the way Internet applications are used today and the underlying infrastructure supporting such applications. Further, we explored ways *Swift* may embed additional ICN properties in its behavior by leveraging existing technologies and infrastructure, such as decentralized peer discovery mechanisms and standard IP routing.

9. ACKNOWLEDGMENTS

We would like to thank Arno Bakker and Pehr Söderman for providing us with valuable feedback. The research leading to these results has received funding from the Seventh Framework Programme (FP7/2007-2013) under grant agreement No. 216217 (P2P-Next).

10. REFERENCES

[1] libswift Homepage. `http://libswift.org`.

[2] PSIRP Project. `http://www.psirp.org`.

[3] The git source code management system. `http://git-scm.com`.

[4] Cisco Visual Networking Index: Usage Study. `http://cisco.com`, October 2010.

[5] R. Allbery and C. Lindsey. RFC 5537: Netnews Architecture and Protocols.

[6] S. Arianfar, P. Nikander, and J. Ott. On content-centric router design and implications. In *ReARCH'10*.

[7] S. Arianfar, J. Ott, L. Eggert, P. Nikander, and W. Wong. A Transport Protocol for Content-Centric Networks. In *ICNP'10. Poster Session*.

[8] A. Bakker. Merkle hash torrent extension, BEP 30. `http://bittorrent.org/beps/bep_0030.html`.

[9] G. Carofiglio, M. Gallo, and L. Muscariello. Bandwidth and storage sharing performance in information centric networking. In *SIGCOMM ICN'11*.

[10] B. Cohen. Incentives Build Robustness in BitTorrent, 2003.

[11] C. Dannewitz, J. Golic, B. Ohlman, and B. Ahlgren. Secure Naming for a Network of Information. In *INFOCOM'10*.

[12] B. Ford, P. Srisuresh, and D. Kegel. Peer-to-Peer Communication Across Network Address Translators. `http://www.brynosaurus.com/pub/net/p2pnat/`.

[13] A. Ghodsi, T. Koponen, J. Rajahalme, P. Sarolahti, and S. Shenker. Naming in Content-Oriented Architectures. In *SIGCOMM ICN'11*.

[14] V. Grishchenko. The Generic Multiparty Transport Protocol (swift). `draft-grishchenko-ppsp-swift-03.txt`.

[15] 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.

[16] V. Grishchenko and J. Pouwelse. Binmaps: hybridizing bitmaps and binary trees. PDS Technical Report PDS-2011-001.

[17] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *CoNEXT '09*.

[18] 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.

[19] 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*, pages 1–2. IEEE, 2010.

[20] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A data-oriented (and beyond) network architecture. In *SIGCOMM '07*.

[21] R. C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *CRYPTO*, 1987.

[22] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind. Low Extra Delay Background Transport (LEDBAT). `draft-ietf-ledbat-congestion-04.txt`.

[23] W. Wong and P. Nikander. Secure naming in information-centric networks. In *ReARCH '10*.

[24] L. Xiao, D. A. Bryan, Y. Gu, and X. Tai. A PPSP Tracker Usage for Reload. `draft-xiao-ppsp-reload-distributed-tracker-03.txt`.

[1] Real-time estimation available at `http://dsn.tm.uni-karlsruhe.de/english/2936.php`

Paper E

Chapter 10

Integrating Smartphones in Spotify's Peer-Assisted Music Streaming Service

R. Jimenez, G.Kreitz, B. Knutsson, M. Isaksson, S. Haridi

Submitted for publication.

Integrating Smartphones in Spotify's Peer-Assisted Music Streaming Service

Raul Jimenez*, Gunnar Kreitz*†, Björn Knutsson*, Marcus Isaksson† and Seif Haridi*
*KTH - Royal Institute of Technology, Stockholm, Sweden
†Spotify, Stockholm, Sweden

***Abstract*—Spotify is a large-scale peer-assisted music streaming service. Spotify's P2P network serves 80% of music data to desktop clients. On the other hand, the rapidly growing number of mobile clients do not use P2P but instead stream all data from Spotify's servers.**

We enable P2P on a Spotify mobile client and empirically evaluate the impact of P2P protocols (in particular low-bandwidth traffic between peers) on energy consumption, both on 3G and Wifi. On 3G, current P2P protocols are highly energy inefficient, but simple modifications bring consumption close to the client-server configuration. On Wifi, the extra energy cost of enabling P2P is much lower.

Finally, we propose a protocol modification to further integrate mobile devices in Spotify's P2P network according to their capabilities (power source, access network). This allows us to break the artificial division between desktop and mobile platforms and dynamically adapt as resources become (un)available to the device.

I. Introduction

While Peer-to-peer (P2P) technology has enjoyed great success for content delivery over the "regular" Internet (wired and Wifi), this is currently not the case for the mobile Internet and the rapidly increasing population of smartphones. In this paper, we challenge the conventional wisdom regarding P2P on such devices, and using the P2P network and clients of the popular Spotify music service, we demonstrate that doing so can lead to significant benefits, while incurring few, if any, of the dire consequences predicted.

Spotify is a *peer-assisted* music streaming service, with over 20 million tracks and over 24 million users in 28 countries.

Currently, Spotify uses its P2P network as a major source of the music played by its desktop client (80% of the total volume), while its own servers primarily handle cases where the P2P network is unable to provide data in a timely fashion to a client.

On its *mobile platforms* (Android, iOS, Windows Phone, etc.) this it not the case, however. These clients follow conventional wisdom, and get 100% of their data directly from Spotify's servers. This situation is not unique to Spotify —we are not aware of any large-scale P2P system where mobile platforms are well integrated— despite the dramatic increase in use of such devices.

At first glance, conventional wisdom appear to be well-founded. P2P systems are predicated on peers contributing "spare" resources for the greater benefit of all, so trying to include a peer which is severely resource constrained (limited battery, storage, CPU and bandwidth, the latter often even capped and metered) may seem quite unwise, and we agree.

Our contention is with the application of it, which we believe to often be too broad. Sure, contributing resources from a mobile phone with those limitations is probably a bad idea, but just letting it *download* from other peers should be no different than having it download from a server.

And what about when that mobile is plugged into the charger and using the Wifi network at home? Then its limitations are no different than those of a laptop, so why can't we let it upgrade itself to a "full" peer then?

In this paper, we will explore these and other related questions and issues by way of trying to let Spotify's client for mobile devices use and participate in the Spotify P2P network. Along the way, we will also examine the P2P protocols and the power consumption they are responsible for on both 3G and Wifi wireless networks, in the hope that by better understanding the causes, it will be possible reduce power consumption.

We observe that power consumption is determined not only by traffic volume, but also by its shape. In wireless networks there are high fixed costs for starting transmission and low marginal costs for additional bytes once the radio interface is powered up. This means that protocols continuously exchanging small amount of data are very power inefficient. Unfortunately, that is exactly the traffic pattern many P2P protocols (including Spotify's) exhibit.

We start from Spotify's desktop client, compiled to run on a mobile system, and from examining the power used on both Wifi and 3G, we explore ways to reduce, sometimes drastically, the power consumed by it. Our modifications include preventing the device from uploading to reducing the "trickling" traffic, and we end up with energy consumption is comparable to the non-P2P configuration.

We also discovered that on Wifi, the increase in energy consumption due to "trickling" traffic is much smaller than we had expected, based on related work [7], [9].

We conclude by proposing guidelines and methods for how mobile devices can be integrated into P2P-systems with a minimum of added power consumption when the systems are battery powered and using 3G, and with most, if not all, the benefits of "full" peers when they are AC powered and on Wifi networks.

We believe, based on our experience with other P2P protocols and networks, that the majority of our conclusions and

proposals are applicable not only to Spotify's P2P network, but to varying degrees, to all the P2P protocols and networks currently in wide use.

We also believe that getting mobile systems integrated into these P2P networks is of paramount importance if P2P is to stay relevant, given the rapid growth of mobile devices population and the ways they are replacing traditional (desktop) systems.

A. Our Contributions

In this work, we make a case study of the feasibility of integrating mobile devices into a deployed, large-scale, peer-assisted system. In this study, we make several finds on energy consumption patterns, showing the surprising efficiency of the unmodified Spotify P2P protocol on Wifi. We also demonstrate how simple backwards-compatible adaptions can drastically reduce energy consumption on 3G.

Our case study leads us to a discussion which could serve as a foundation for future research: how can overlay designs best be updated to accomodate the new class of mobile devices? These devices have new characteristics as participants in peer-to-peer overlays as they can up-front be flagged as resource-constrained, but may also swiftly change status when on Wifi and connected to a power supply.

Lastly, the tools we developed for this study have been released as open source, available at http://people.kth.se/~rauljc/spotify/. We believe these may be of independent use to analyze and understand power consumption of networked applications. In particular, some potential energy optimizations for the current (client-server) Spotify protocol were discovered using these tools.

II. Spotify and P2P

This section provides a brief overview of Spotify and its peer-assisted mechanisms. See [1], [2] for more technical details.

The core compontent in Spotify's service is music delivery. Clients can request music tracks from both Spotify's servers and its P2P network. At present only desktop clients use the P2P network, and the mobile clients only request data from Spotify's servers.

Desktop clients prefer downloading tracks from the P2P network. They only stream from the server to guarantee low-latency and gap-less playback. For instance, when a user selects and plays a track, the server provides the beginning of the track while the client requests the remaining data from the P2P network. When playing from a playlist, the client requests the next track from the P2P network a few seconds before the current track ends, downloading the whole track from the P2P network in the best case. In any case, the client can always fall back to requesting data from the server to prevent music interruptions and stutter.

To minimize the amount of data delivered by servers, a client requests data in streaming mode, avoiding requests more than 15 seconds ahead of the current playback point. On the P2P network, whole tracks are downloaded as fast as possible.

On mobile platforms, where P2P is not available, clients request tracks in large chunks: an initial request of around 1.5 MB and then requests of up to 5 MB. The reason behind this streaming strategy is a trade-off attempting to minimize both energy consumption and delivery of unnecessary data. We will discuss energy consumption in Section III.

A. Peer Discovery

Spotify clients use two different peer discovery mechanisms: 1) a centralized tracker and 2) a distributed overlay. A design rationale for having two mechanisms is that it allows both to be imperfect and at times miss peers, making them more efficient and cheaper to implement. These two mechanisms have similarities to those of BitTorrent and Gnutella, respectively.

The tracker service is implemented in Spotify's backend. Compared to BitTorrent's tracker, an important difference is that a Spotify client will only announce itself to the tracker after it has downloaded a complete copy of the music track. That is, Spotify's tracker only tracks seeders. Spotify's tracker also limits the amount of peers it tracks and the number of responses to queries to keep overhead low.

The second mechanism is an unstructured overlay similar to Gnutella's and is very relevant to this paper, since its networking patterns greatly affect energy consumption on mobile devices.

To form the overlay, every peer establishes and maintains a number of TCP connections to other peers (called neighbors), the details of how neighbors are selected can be found in [1], [2].

To find peers offering a given track, a client sends a query to all its neighbors and these neighbors in turn forward the query to their neighbors. That is, all peers at most two hops away from the client will receive the query and respond if they happen to have the requested track.

On the other end, each peer constantly receives queries and keep-alive pings from neighbors. In terms of peak bandwidth and total number of bytes, the overhead is relatively small compared to data transfers. On wireless networks, however, continuous low-bandwidth traffic is considered expensive in terms of energy consumption. We will discuss energy consumption in wireless networks in Section III.

III. Energy Consumption in Wireless Networks

One of the main concerns for users and smartphone app developers is battery life. The proliferation of battery-hungry apps make smartphones' batteries last shorter and the availability of energy monitors give users the tools to identify and uninstall apps that provide low value per joule.

Several studies have investigated energy consumption on mobile devices. Zhang et al. [3] and Yoon et al. [4] studied in detail power consumption caused by the different hardware components of a smartphone, including Wifi and 3G.

In this paper, we focus on power consumption caused by network activity in Wifi (IEEE 802.11g) and 3G. While a brief description of their power properties is given in this section, we refer to the studies above for further details.

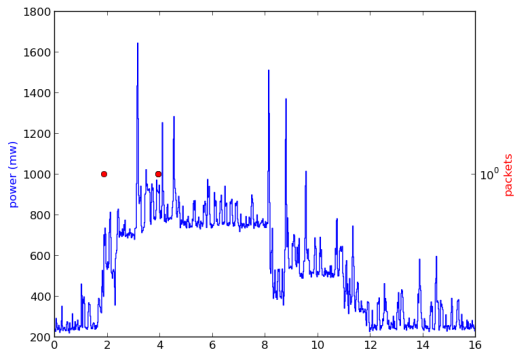

Fig. 1. A ping-response exchange shows power transitions on 3G: DCH (2-8) and FACH (8-12). Notice the latency of around 2 seconds.

A. 3G

There are three power states in the 3G radio resource controller (RRC): IDLE, FACH, and DCH. To transmit data, the radio needs to be powered up to DCH where the device acquires a dedicated channel for communication. After a few seconds of inactivity, the radio is demoted to FACH, where it is assigned a shared channel. In FACH, power consumption is about half of DCH and the state can be promoted to DCH quickly if more data needs to be transmitted. After a few further seconds of inactivity, the radio is powered down to IDLE where it consumes very little power.

Figure 1 illustrates 3G power states on an Android smartphone. Approximately two seconds into the graph, the phone sends a TCP packet to a server (red dots represent packets captured by tcpdump). To transmit the packet, the 3G radio transitions into DCH, increasing power consumption. Around two seconds, the packet reaches the server, which sends a TCP ACK back. Finally, about four seconds into the graph, the smartphone receives the TCP ACK.

This example illustrates the long initial delay (actual RTT was below 100 ms) applications are subjected to. Precisely, to counter such long delay for subsequent packets, the interface stays in DCH for around four seconds and an extra four seconds in FACH before it powers down back to IDLE. Inactivity timeouts are controlled by network operators and can vary widely.

B. Wifi

Wifi data transfer is much more efficient than 3G, specially when power saving mechanism (PSM) is enabled. PSM lets the radio interface sleep and just wake up at regular intervals to listen for beacons from the access point. These beacons indicate whether the mobile device has received data. This data is buffered by the Wifi access point and can be retrieved by the client when it wakes up.

While PSM adds latency, it saves energy. Modern smartphones support adaptive PSM (PSM-A) which keeps the wireless interface powered for an extra period of time to lower latency and increase throughput. In modern smartphones this extra period (i.e. energy tail) has been estimated to be approximately 200 ms [5].

A detailed description of Wifi power consumption can be found in [6]. A recent survey on energy-efficient streaming further elaborates on power saving methods in Wifi [5].

IV. Experimental setup

To measure and understand the sources of power consumption, we have run experiments in different networks and locations. While details are slightly different in each experiment, all experiments show the same patterns. Given that we are interested in these patterns rather than details, we consider the experiments we document in this paper to be representative.

We run each experiment sequentially on a Galaxy Nexus phone running Android 4.2.1. For each Spotify client configuration (see next section for details), we instruct the app to play a playlist for 31 minutes. The playlist played is *Top 100 tracks currently on Spotify*[1] as of June 20th 2013. We chose this playlist because its tracks are well seeded, making sure that the mobile client would find enough peers to download all data from the P2P network when P2P was enabled.

Spotify uses the Ogg Vorbis format for streaming in three different quality alternatives: q3 (96 kbps), q5 (160 kbps), and q9 (320 kbps). The defaults are q3 and q5 for mobile devices and desktop clients, respectively.

To be able to download data from existing desktop clients, we select *High quality* (q5) on the mobile device. We also make sure that Facebook reporting and last.fm scrobbling were disabled so that communication with these services would not interfere with our experiments. For the same reason, we use a phone exclusively for these experiments with no extra apps installed and background synchronization disabled.

In addition, we check all connections the client makes in each experiment. Despite our efforts to eliminate non-Spotify traffic from the experiments, we do observe long-lived connections to Google's servers (*.1e100.net) in which small amounts of data are exchanged regularly.

Each experiment starts with an empty cache and the first minute is discarded, to exclude user interaction when the display is powered. During the 30 minutes we analyze, the display is off and headphones are plugged in. Volume is set to the lowest setting (volume setting does not affect power consumption [3]).

Each configuration is evaluated both on 3G and Wifi. For the entire duration of the experiments the smartphone is stationary. We use an enterprise 3G package from Swedish operator Telia with no monthly data caps. According to Android, the mobile network types used are: HSDPA:8, HSPA+:15, and UMTS:3. On Wifi, we report the experiments run on the Wifi

[1] http://play.spotify.com/user/spotify/playlist/4hOKQuZbraPDIfaGbM3lKI

infrastructure provided by KTH in its Kista campus. We have also repeated the experiments at Spotify's Stockholm offices with no major differences in the results, despite the fact that the former provides public and the latter private IP addresses.

To measure power consumption, we use a Monsoon Power Monitor[2], which feeds power at constant voltage to the phone while collecting up to 5000 power samples per second. During recalibration periods, the device fails to report samples. We compensate these missing samples by reducing the sampling frequency. For instance, 50 samples per second where used in Figure 1. That is, every 20 ms our tools output a single aggregate sample by averaging all samples reported by the device in that period of time. In the next section, we show figures using one aggregated sample per second (1 Hz) to better illustrate power patterns on 3G by removing noisy power spikes.

To capture network traffic on the mobile device, we run *tcpdump* provided by Shark for Android[3].Since tcpdump requires root privileges, we rooted the phone using Wug's Nexus Root Toolkit v1.6.2[4].

Our tools analyze the traffic capture files, generating useful information such as per host traffic analysis as well as the graphs included in this paper.

While these tools have been created specifically for this project, they are designed to be generic. In fact, they are used by Spotify to identify and remove spurious data traffic from its production client-server mobile client. These tools are written in Python and are available under an open-source license at http://people.kth.se/~rauljc/spotify.

V. Experimental Results

A. Baseline: P2P Disabled

The first configuration we evaluate is the standard Spotify app[5], which we use as *baseline* for comparison. We compile the unmodified source code for version 0.6.0 and manually install the Android package on our test mobile device. We repeat the same process for the other two configurations evaluated in this section.

This *baseline* configuration does not use P2P, instead downloading all data directly from Spotify's servers. The smartphone client makes fewer and larger requests compared to its desktop counterpart, following recommendations for Android developers[6]. A primary reason for this recommendation is that it is much more energy efficient to download the whole track as fast as possible than performing long low-bandwidth transfers on wireless networks. We refer the reader to a recent survey [5] for an in-depth study of energy-efficient multimedia streaming on wireless networks.

[2]http://www.msoon.com/LabEquipment/PowerMonitor/

[3]https://play.google.com/store/apps/details?id=lv.n3o.shark

[4]http://forums.androidcentral.com/verizon-galaxy-nexus-rooting-roms-hacks/147177-automated-wugs-nexus-root-toolkit-v1-6-2-updated-12-29-12-a.html

[5]https://play.google.com/store/apps/details?id=com.spotify.mobile.android.ui

[6]http://developer.android.com/training/efficient-downloads/efficient-network-access.html

Baseline downloads each music truck in chunks. An initial download of 1.5 MB, followed of up to 5 MB chunks. This simple heuristic attempts to save power while avoiding premature download of data that will not be used if the user skips the track a few seconds into playback, which is common user behavior.

Figure 2 shows a typical track playback. The figure shows power consumption (blue line) and the number of IP packets transmitted within one second (red dots, log scale). Power is shown in one-second sample aggregation (as explained in Section IV) to remove noise and emphasize power patterns. This visualization helps us to clearly see the impact of network traffic on power consumption.

The track is downloaded in two chunks. At second 39, 1.5 MB are downloaded, the rest of the track (3.7 MB) is downloaded at second 100. In the figure, these downloads are clearly identifiable by a cloud of dots above 300 packets/s.

The 3G energy tail is clearly seen and it lasts several seconds. It is also worth noting that it takes around two seconds between the client's request for content and the first data packet from the server, this time is spent negotiating and setting up the channel. The phone's 3G radio is active during these two seconds. In the figure, this is visible as a red dot indicating one packet just before the download.

The figure also shows periods of low network activity where a few packets are exchanged triggering a long period of high power consumption. The worst offender is found at second 212. A 3-packet exchange keep-alive ping (sent 11 bytes, received 11 bytes, ACK) between the client and the server keeps the 3G radio interface up for around 10 seconds.

Similar patterns are triggered by notifications and reporting messages. For instance, to report that a particular track has been played.

Despite our efforts to eliminate non-Spotify traffic from the experiments, we also observe a DNS query plus a 17-packet exchange (4202 bytes) with Google's servers (arn06s02-in-f19.1e100.net) at 170 s.

In this particular example, downloading periods (including startup delay and tail) consume 32 joules in 39 seconds. Low-bandwidth traffic consumes 43 J in 68 s and while low-power periods consume 41 J in 143 s. This means that bundling the low-bandwidth traffic together with a download would significantly reduce energy consumption.

This optimization is out of the scope of this paper. Although, as a result of this research, Spotify will modify its protocol to reduce power consumption on mobile devices.

B. Peer: Unmodified P2P

Next, we enable P2P on the mobile device with the same configuration as the Spotify desktop client. Although this configuration is capable of uploading data to other peers, there is no data upload in our experiments, allowing us to study solely download traffic and P2P overhead.

Compared to baseline, there are two major differences with respect to traffic patterns: single download and continuous low-bandwidth traffic.

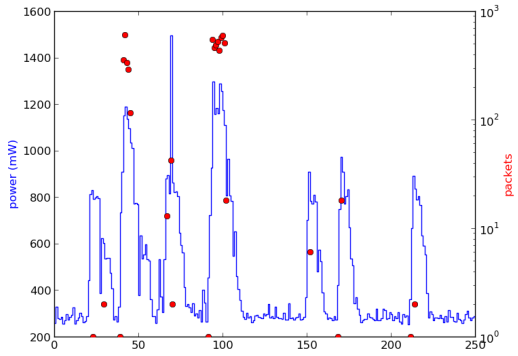

Fig. 2. Baseline 3G. Track download is done in two bursts. Notice energy end tail. Power sampling: 1 Hz.

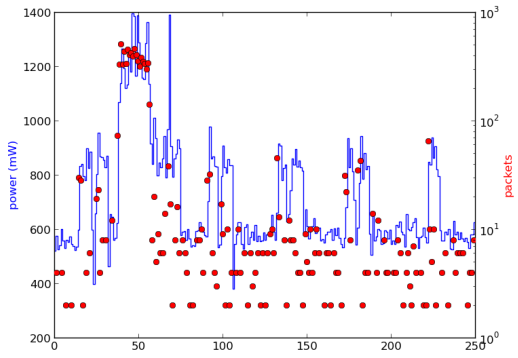

Fig. 3. Peer 3G. Track download is done in one burst. Energy end tail keeps 3G constantly awake, consuming extra power. Power sampling: 1 Hz.

Figure 3 shows a single large dot cloud (around second 50), which was expected, since downloads from the P2P network are always done in a single chunk as described in Sectionbackground-spotify. While this is more energy efficient, it risks downloading useless data if the user skips the track.

As expected, Spotify's Gnutella-like query flooding mechanism (used to find peers) generates a continuous flow of low-bandwidth traffic, keeping the 3G radio interface powered almost continuously, dramatically increasing energy consumption. Over a 30 minute period, this configuration almost doubles energy consumption compared to baseline, as shown in Section V-D.

Figure 4 shows the same configuration on Wifi. Notice that we increased aggregation sampling to 50 Hz in the power graph to show power spikes, given the much shorter tail energy

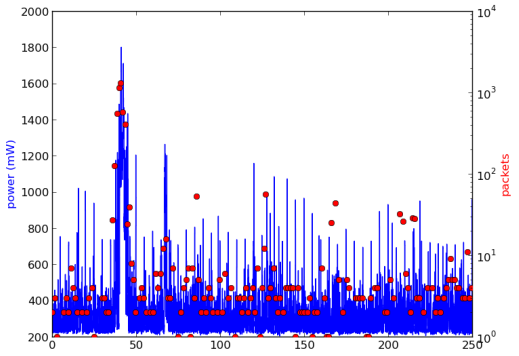

Fig. 4. Peer Wifi. Music download is done in one burst. Energy end tail is not that bad in Wifi. Power sampling: 50 Hz.

period compared to 3G.

We observe that Wifi is much more power-efficient handling low-bandwidth traffic. It is also much better compared to a 2008 study evaluating constant low-bandwidth traffic generated by another P2P protocol [7]. We speculate this improvement comes from Wifi power saving methods implemented in modern devices.

C. Leecher: Reducing Energy Consumption

In some contexts, P2P upload from mobile devices is very expensive. For instance a battery-powered device on 3G data plan with a scarce monthly allowance.

We can completely disable upload on mobile devices. This makes mobile peers free riders or *leechers*. Leechers decrease scalability because a leecher demands resources from the system while providing none. On the other hand, in peer-assisted systems, they can download data from other peers, offloading servers.

Leeching can be done by simply refusing uploading to peers requesting data. A better approach is, however, to never register on the peer discovery services, so other peers do not connect the leecher to request data at all.

As described in Section II-A, Spotify has two complementary peer discovery mechanisms: centralized tracker and decentralized flooding search.

Standard peers announce files to the centralized tracker as soon as they are completely downloaded and stored in cache. We modified leechers to use the tracker to find peers but never announce files.

The distributed search is a more complex mechanism. Each peer keeps connections to a number of neighbors. Whenever it needs to find peers, a peer queries all its neighbors, which will 1) reply if they have the file and 2) forward the query to all its neighbors.

We modify the configuration to never reply to queries (no reply means "I don't have the requested data"), and drop

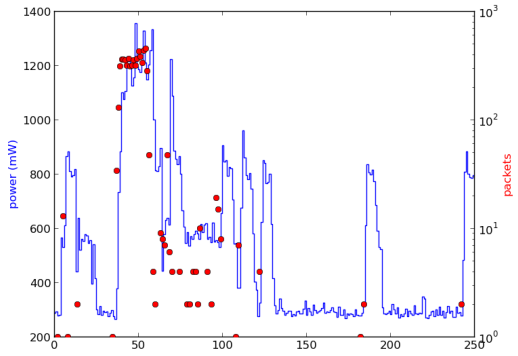

Fig. 5. Leecher 3G. Music download is done in one burst. Energy end tail keeps the 3G radio in a non-idle state for 60 seconds, which is what it takes to drop a neighbor.

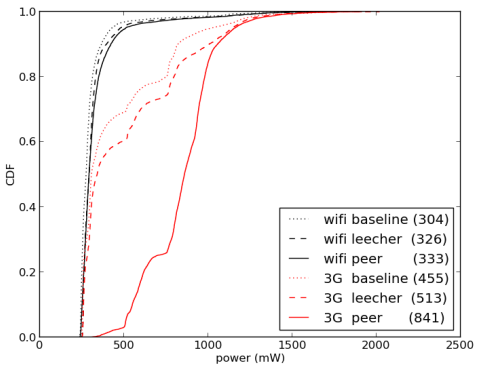

Fig. 6. Power consumption CDF. Legend shows average power consumption (mW).

connections after 60 seconds. While this approach has its drawbacks, it clearly shows the impact of low-bandwidth traffic patterns produced by some P2P protocols on energy consumption. We will later propose improvements to this configuration.

Figure 5 shows that the bulk of low-bandwidth traffic disappears by dropping P2P neighbor connections, dramatically reducing energy consumption while the whole track is downloaded from the P2P network.

A more aggressive dropping policy would further reduce energy consumption at a risk of affecting search performance and ability of downloading data from peers.

While this approach has the benefits of using the distributed search to find peers and being backwards compatible, it has some drawbacks. For instance, it introduces peers that are not using the protocol as originally intended and artificially increase churn[7] by dropping connections after 60 seconds.

Ideally, leechers should be able to find peers using distributed search while not being constantly queried by other peers. One simple solution, which we propose for future work, is to label connections to leechers in a way that peers never send queries through these connections. This would stop the energy-inefficient trickling traffic, while providing distributed search for leechers. While we elaborate on this proposal in Section VI-B, we could not evaluate it empirically because it is backwards incompatible.

D. Total Energy

Figure 6 shows a comparison of the amount of power consumed by each configuration over the 30-minutes experiments, allowing comparison between different client and network configurations. Average power consumption is included in the graph's legend.

[7]Rate at which peers join/leave a P2P overlay.

Starting at the bottom left, we observe that all lines but one are bundled together between 250 and 350 mW. This power consumption corresponds to playback and output audio to headphones while the wireless radio is idle.

All three configurations tested on Wifi (black lines) consume less than 350 mW more than 90% of the time. There is a relatively small power gap between baseline and the other two configurations, showing that modern mobile devices can efficiently handle low-bandwidth traffic on Wifi, minimizing the impact of inefficient Wifi on P2P overlays found in the literature (e.g. [7]).

On 3G, we can clearly see a staircase-shaped line which corresponds to 3G's power states and their long inactivity timeouts. In the extreme case of the *peer* configuration, the 3G radio is constantly powered in a non-idle state and spends most of the time in DCH (highest power state). As discussed before, this situation is caused by a constant flow of low-bandwidth traffic generated by the distributed search protocol.

We also observe that the simple modification of dropping P2P connections after 60 seconds is enough to greatly decrease energy consumption. A more aggressive dropping policy would further reduce the gap between baseline and leecher.

These results suggest that it is possible to integrate mobile devices into P2P, at least as leechers. Even with a naive implementation of a leecher, the energy cost in Wifi is moderate. On 3G, the gap between baseline is much larger but we were able to greatly reduce it with simple backwards-compatible protocol modifications. In the next section, we propose further improvements, introducing backwards-incompatible modifications.

VI. Moving Forward

A. Peers and Leechers

Until now, we have hinted at how to categorize mobile devices into peers and leechers. Here, we propose a heuristic

that we consider appropriate for Spotify.

Ideally, participants in the P2P network should be categorized according to their available resources. That would partially eliminate an artificial distinction between devices and platforms (operating systems). And, more importantly, it would allow switching from one category to another as resources become available.

Our experiments have focused on energy consumption because battery life span is a major issue on mobile devices. But not all mobile devices are battery-powered all the time. Many users charge their devices at night, while at work or driving. Many devices are docked into sound systems and there are already docking stations in the market that convert a smartphone or tablet into a full-fledged desktop computer with all necessary peripherals.

Another factor is data caps. Currently, most mobile data plans offered by operators have data caps. A common deal is a monthly allowance of 0.5 to 2 GB, including both up- and down-stream traffic. In fact, many applications consider this limitation, limiting large downloads (e.g. Spotify's off-line synchronization) to be done only on Wifi, by default.

Given these two factors, we consider as *peers* those mobile devices connected to a power supply and using Wifi. Given the widespread use of data caps on mobile networks and the lack of a standard way to automatically detect these caps, we consider all mobile devices connected to a mobile network as *leechers*, regardless of their power status.

B. Protocol Modification

In our experiments, we tried to create two peer categories: peer and leecher. Unfortunately, we could not evaluate a true leecher configuration due to backward incompatibility issues. That is, the leecher received queries from existing peers and the only way to stop them was dropping peer connections which, in turn, artificially increased churn.

In this section, we propose a backwards incompatible modification to the P2P protocol modification that not only addresses this problem but it also tackles seamlessly switching between peer and leecher according to the resources available to the mobile device at any moment.

While deploying backwards incompatible modifications into a large-scale P2P network is hard, we believe that the benefits of integrating mobile devices will compensate the deployment costs.

The modification affects the way peers establish connections to neighbors and how these connections are used. Currently, all peers are considered equal and they frequently query each other to find peers and request data. They also send keep-alive pings to check that the neighbor is still alive (it is common for P2P protocols to assume that peers are unreliable).

Our modification introduces a flag where a participant in the P2P network indicates to its neighbors whether it is a peer or a leecher. Connections to neighbors flagged as leechers are never used to send queries or keep-alive pings. In addition, leechers never announce themselves to the central tracker. That is, leechers cannot be discovered by any of the two peer discovery mechanisms, thus no peer will try to establish a connection to a leecher.

As a result, each leecher will be connected to a number of neighbors (all of them full peers). Whenever a leecher needs to download a track, it will query its neighbors until it finds a peer offering the requested data.

Peers will have neighbor connections to both peers and leechers. The only difference with the current protocol is that peers will not send nor forward queries to leechers.

Notice that these modifications not only stop energy-hungry low-bandwidth traffic but also shields the P2P network from artificial churn. In the modified protocol, a leecher going off-line does not affect the overlay because no message is ever routed through leechers.

This design lets switching categories without the need of a disconnection plus a full bootstrap into the P2P overlay. If a peer becomes a leecher (e.g. the device is disconnected from a power supply), it will drop its neighbor connections to leechers and notify its peer neighbors of its new status, quickly stopping all incoming queries. In the opposite case where a leecher becomes a peer, the status notification will allow both incoming queries from existing neighbors and incoming connections from both peers and leechers.

Switching networks (IP address) is a special case and may, but not necessarily, involve switching categories as well. Provided each device has an identifier independent of its network address (as it happens in most P2P networks), it can reconnect to all its neighbors. The neighbors will update the new IP address and category.

VII. Related Work

There are a number of studies investigating the integration of mobile devices in P2P systems. Nurminen and Nöyränen [8] measured power consumption of SymTorrent—a BitTorrent client for Symbian—both in 3G and Wifi. They report a higher energy consumption on 3G, which is mainly caused by lower download speed on 3G, thus keeping the radio powered for a longer period of time. They also measure "full peer" versus "client only" but only on Wifi.

Kelényi and Nurminen [7], [9] measure energy consumption of Mainline DHT, BitTorrent's decentralized peer discovery mechanism. They highlight the high energy cost of continuous low-bandwidth traffic on Wifi. They reduce energy consumption by selectively dropping DHT queries.

A recent study by Nurminen [10] evaluates energy consumption of BitTorrent and BitTorrent's Mainline DHT. While it studies energy consumption using a Nokia device on a 3G network, it focuses on how BitTorrent's incentives affect download time, and thus energy consumption.

Peer-to-Peer Streaming Protocol (PPSP) is at late stage in the IETF standarization process[8]. Petrocco et al. [11] evaluated the performance of Libswift (reference implementation of PPSP) on Android. The study includes power comparison, using hardware measurements, of Libswift and YouTube playing

[8]http://datatracker.ietf.org/wg/ppsp/charter/

a video stream, using hardware measurements. They did not consider peer discovery, focusing on streaming on Wifi.

RapidStream [12] is P2P video streaming prototype on Android. The paper focuses on design and practicalities with the Android platform. Superficial energy measurements (phone's battery indicator) were given in the paper.

There is a considerable number of design papers and network performance and mobility issues on mobile P2P but do not address energy consumption. For instance, Eittenberger et al. [13] evaluate BitTorrent on WiMax and Berl and Meer [14] consider incentives when integrating mobile devices as leechers in the eDonkey network.

VIII. Conclusion

In our efforts to integrate mobile devices into Spotify's P2P system, we have studied the issues that difficultate their integration.

We make the case against treating mobile devices as resource-constrained devices, but instead categorize them according to the resources they have available. This includes seamlessly switching categories as resources become (un)available to the device.

In particular, we consider two categories: peer and leecher. The peer category is reserved for devices that are capable of performing all tasks expected of a full peer (download and upload data, provide peer discovery service) at low cost to the client in terms of energy and bandwidth. The rest of the devices are considered leechers and are able to download data from the P2P network (offloading servers) with no, or minimal, increase of energy and bandwidth consumption.

Our experiments show that low-bandwidth traffic used by many P2P systems is particularly energy-inefficient on 3G. On Wifi, on the other hand, this traffic does not significantly increase energy consumption. Presumably, this is due to energy saving improvements implemented on modern mobile devices.

We also show in our experiments that a simple backwards-compatible modification goes a long way reducing energy consumption on 3G.

Finally, we propose a protocol modification that would further reduce energy consumption for leechers while allowing them to be integrated in the P2P network. This modification is backwards-incompatible, which is the reason why we have not empirically evaluated it.

Additionally, the tools we developed for this study have been released as open source, available at http://people.kth.se/~rauljc/spotify/. We believe these may be of independent use to analyze and understand power consumption of networked applications. In particular, some potential energy optimizations for the current (client-server) Spotify protocol were discovered using these tools.

Disclaimer

We would like to emphasize that this is an academic research paper and should not be taken as indicative of Spotify's product plans.

Acknowledgment

We thank Spotify employees, in particular Javier Ubillos and his team, who not only provided technical support and were involved in the research discussions, but also provided a great working environment.

We also thank engineers from Telia and Deutsche Telekom who helped us understanding the problem from an operator point of view.

References

[1] G. Kreitz and F. Niemela, "Spotify – large scale, low latency, p2p music-on-demand streaming," in *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on*, 2010, pp. 1–10.

[2] M. Goldmann and G. Kreitz, "Measurements on the spotify peer-assisted music-on-demand streaming system," in *Peer-to-Peer Computing (P2P), 2011 IEEE International Conference on*, 2011, pp. 206–211.

[3] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, ser. CODES/ISSS '10. New York, NY, USA: ACM, 2010, pp. 105–114. [Online]. Available: http://doi.acm.org/10.1145/1878961.1878982

[4] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha, "Appscope: Application energy metering framework for android smartphone using kernel activity monitoring," in *USENIX ATC*, 2012.

[5] M. Hoque, M. Siekkinen, and J. Nurminen, "Energy efficient multimedia streaming to mobile devices — a survey," in *IEEE Communications Surveys & Tutorials*. IEEE, 2013.

[6] Y. Xiao, P. Savolainen, A. Karppanen, M. Siekkinen, and A. Ylä-Jääski, "Practical power modeling of data transmission over 802.11g for wireless applications," in *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, ser. e-Energy '10. New York, NY, USA: ACM, 2010, pp. 75–84. [Online]. Available: http://doi.acm.org/10.1145/1791314.1791326

[7] I. Kelenyi and J. Nurminen, "Optimizing energy consumption of mobile nodes in heterogeneous kademlia-based distributed hash tables," in *Next Generation Mobile Applications, Services and Technologies, 2008. NGMAST '08. The Second International Conference on*, 2008, pp. 70–75.

[8] J. Nurminen and J. Noyranen, "Energy-consumption in mobile peer-to-peer - quantitative results from file sharing," in *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, 2008, pp. 729–733.

[9] I. Kelenyi and J. Nurminen, "Energy aspects of peer cooperation measurements with a mobile dht system," in *Communications Workshops, 2008. ICC Workshops '08. IEEE International Conference on*, 2008, pp. 164–168.

[10] J. Nurminen, "Energy efficient distributed computing on mobile devices," in *Distributed Computing and Internet Technology*, ser. Lecture Notes in Computer Science, C. Hota and P. Srimani, Eds. Springer Berlin Heidelberg, 2013, vol. 7753, pp. 27–46. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36071-8_3

[11] R. Petrocco, J. Pouwelse, and D. H. J. Epema, "Performance analysis of the libswift p2p streaming protocol," in *Peer-to-Peer Computing (P2P), 2012 IEEE 12th International Conference on*, 2012, pp. 103–114.

[12] P. Eittenberger, M. Herbst, and U. Krieger, "Rapidstream: P2p streaming on android," in *Packet Video Workshop (PV), 2012 19th International*, 2012, pp. 125–130.

[13] P. M. Eittenberger, S. Kim, and U. R. Krieger, "Damming the torrent: Adjusting bittorrent-like peer-to-peer networks to mobile and wireless environments," *Advances in Electronics and Telecommunications*, vol. 2, no. 3, pp. 14–22, 2011.

[14] A. Berl and H. de Meer, "Integrating mobile cellular devices into popular peer-to-peer systems," *Telecommunication Systems*, vol. 48, no. 1-2, pp. 173–184, 2011.

Paper F

Chapter 11

Tribler Mobile: P2P Video Streaming from and to Mobile Devices

R. Jimenez, A. Bakker, B. Knutsson, J. Pouwelse, S. Haridi

Submitted for publication.

Tribler Mobile: P2P Video Streaming from and to Mobile Devices

Raul Jimenez*, Arno Bakker†, Björn Knutsson*, Johan Pouwelse†, Seif Haridi*
* KTH - Royal Institute of Technology, Stockholm, Sweden
† Delft University of Technology, Delft, The Netherlands

***Abstract*—Peer-to-peer (P2P) mechanisms allow users with limited resources to distribute content to a large audience, without the need of intermediaries.**

These P2P mechanisms, however, appear to be ill-suited for mobile devices, given their limited resources: battery, bandwidth, and connectivity. Even Spotify, a commercial straming service where desktop clients stream about 80% of the data via P2P, does not use P2P on mobile devices.

This paper describes Tribler Mobile, a mobile app that allows users to broadcast their own videos to potentially large audiences directly from their devices. Our system delegates most of the distribution tasks to *boosters* running on desktop computers. Our mechanisms are designed to be fully-decentralized and consider mobile devices' limitations.

Tribler Mobile is available as open-source software and have been installed by almost 500 users on their Android devices.

I. Introduction

This paper presents a P2P video streaming platform where mobile devices are well integrated. As a prove of concept, we have modified an existing P2P system and deployed the infrastructure to allow mobile devices not only stream from the P2P network, but also publish new content to it.

We make the very conservative assumption that uploading data from mobile devices is always expensive due to cost related to increased energy consumption on battery-powered devices and traffic (monthly data caps are common on mobile networks). In practice, we expect a more favorable environment where uploading costs are much lower for some mobile devices; for instance, when a device is connected to a power supply and on a wired or Wifi connection. Furthermore, more generous data caps and more energy-efficient devices may lower uploading costs on mobile devices in the future.

In this environment, a mobile device publishing new content should ideally upload a single copy of the content to the P2P network, and then immediately stop all P2P traffic. This is basically what cloud-based applications do, and they can do it because cloud servers are reliable and they are always ready to accept data. In P2P networks, however, each peer is assumed to be unreliable and publishers have no easy-to-use mechanisms to place peers on stand-by, waiting for new content.

We are the first to propose, implement, and deploy a complete system where mobile devices can publish content directly into the P2P network and let other peers distribute the content to a potentially large audience (including other mobile devices). In the process of building this system, we develop a reverse peer discovery mechanism (described in Section IV-A) that (1) traverses NAT gateways —these gateways are widely-deployed on Wifi and mobile networks— and (2) limits energy consumption by preventing other peers from establishing connections to the mobile device (those connections would switch the device's radio interface to high-power state even after the P2P app has terminated).

To distribute content to other mobile devices, our P2P system organizes *desktop peers* (peers running on desktop computers) to form a sort of content distribution network (CDN). Some of these desktop peers would be run by users who consume the content, and as a side-effect, contribute to its distribution to both desktop and mobile peers. This paper, however, is mainly focused on what we call *boosters*.

A *booster* is a peer whose main task is to augment (or boost) the distribution capacity of a publisher or source. There could be different reasons why a user chooses to cede resources to a publisher by boosting its content: personal relationship, political/activist support, in exchange for services or money, et cetera.

Boosters are meant to provide the capacity demanded by *leechers*. A leecher is a peer that downloads data from other peers, without uploading data to others[1]. While the term leecher is often used pejoratively to indicate that these leechers are free-riders, we use it without any negative connotation. On the contrary, our system is designed to address the extra capacity demanded by leechers by creating extra capacity provided by boosters.

This paper presents a system design that is generic and flexible enough to be deployed in different contexts. From an open-community best-effort system to a commercial P2P streaming service where boosters are mainly (or exclusively) run by the publisher to guarantee quality of experience. While open-community services tend to demand fully-decentralized solutions, commercial services may keep some subsystems centralized for monitoring and control purposes and to ease deployment.

As a proof of concept, we have implemented and deployed a complete prototype which consists of these elements:

- A mobile app, called *Tribler Mobile*, offers P2P video streaming playback, as well as publishing new content to the P2P network. For instance, a video recorded with the smartphone's camera.

[1]Throughout the paper, content is always streamed, though we use *download* and *upload* to indicate data flow's direction.

- A desktop application, called *Tribler*, which can be configured to automatically boost certain content (e.g., published by a friend) as soon as it is published.[2]
- A subscription and publisher authentication mechanism (Section IV-B). We use Twitter in our prototype in a way when a booster subscribes to `@user`, it will boost all content posted by that user. That is, when `@user` records a video and publishes it, a tweet is automatically posted on `@user`'s timeline and that video will then be boosted by all boosters subscribed to `@user`.
- A DHT-based reverse peer discovery (Section IV-A that addresses two issues at once. When the mobile device (MD) is behind a NAT gateway, it allows the MD to upload the content by establishing direct connections to boosters. When the MD is not behind a NAT gateway, it prevents connection attempts from other peers, which would increase energy consumption even after the mobile app had been terminated.

The rest of this paper is organized as follows. Background is presented in Section II. Section III introduces the roles in the system and how they interact with each other. In Section IV, we describe each of the infrastructure services supporting our system, both in abstract terms and our prototype's concrete implementation details along with a short discussion about trade-offs and alternative implementations. Finally, Section V concludes.

II. Background

Internet content streaming services are very popular and growing fast. Not only on large-screen wire-connected desktops but also battery-powered wireless-connected mobile devices. In fact, all trends indicate that mobile devices already account for a large part of streaming, and it is growing far faster than on desktop platforms. For instance, mobile devices consumed 41% of YouTube's total traffic in the third quarter of 2013, up from 25% in 2012 and just 6% in 2011.[3]

Streaming content to a large audience is far from trivial. In general, it is impractical for an individual or a small company to deploy its own content distribution system. While there are platforms that will distribute your content for a fee (e.g., content distribution networks) or placing ads (e.g., YouTube), there is an alternative: *P2P content distribution*.

A. Peer-to-Peer Content Distribution

On the desktop environment, where desktop computers and laptops are commonly connected to a power source and connected to the Internet though Wifi or a wired network, P2P offers the possibility of augment the publisher's distribution capacity with resources from its viewers. That is, the viewers not only consume resources but also contribute their own, increasing scalability and reducing costs to the publisher.

File-sharing communities use P2P to distribute content by each peer contributing its own resources (mainly storage and bandwidth). P2P systems have evolved to be fully-distributed and self-organized, gradually removing centralized services (e.g., tracking and torrent files acquisition in BitTorrent, which initially relied on centralized services, have now been fully decentralized).

Yet, mobile devices have not been well integrated on P2P networks. Conventional wisdom suggests that these devices are ill-suited for P2P for several reasons: (1) uploading would drain batteries and deplete 3G monthly traffic allowances too fast for most users, (2) connectivity issues caused by widely-deployed network address translator (NAT) gateways, and (3) for some publishers, client-server (e.g., content distribution networks (CDNs)) distribution costs are low enough to disregard investments in P2P-based optimizations.

Spotify, a commercial music streaming service, is a good example to illustrate this point. On desktop platforms, Spotify uses P2P mechanisms to increase scalability and reduce costs. According to their own measurements, 80% of the music is delivered by its P2P network [2]. On mobile platforms, however, all music is currently delivered by Spotify's servers.

B. Mobile P2P

The introduction of Internet-enabled mobile devices led some researchers to investigate the feasibility of mobile devices participating in P2P systems.

Bakos et al. [3], [4] simulated Gnutella on a variety of wireless topologies. Then, Nurminen and Noyranen [5] studied BitTorrent's impact on energy consumption on Symbian-based smartphones on both 3G and Wifi networks, considering it feasible to run full peers on mobile devices (download and upload) since BitTorrent's tit-for-tat mechanism [6] rewards uploaders with faster downloads, thus decreasing energy consumption through a shorter total downloading time. Notice that tit-for-tat is less relevant in seeding-rich environments, like the one we propose in this paper.

Kelényi and Nurminen studied the energy aspects of a mobile devices on BitTorrent's Mainline DHT [7]. They concluded that running a full peer on a mobile device is unfeasible due to continuous incoming queries that keep the radio interface on a high-energy state. On the other hand, their client-only implementation allowed mobile devices to use DHT-based services with minimum energy cost.

In a paper currently under review [8], we explored the participation of mobile devices in Spotify's P2P network, in a way that these devices can stream data not only from Spotify's servers (as it currently happens) but also from other users running desktop Spotify clients. When we enabled P2P on the Spotify's Android app, we found that Spotify's gossip-like peer discovery mechanism increases energy consumption (moderately on Wifi and dramatically on 3G). After applying backwards-compatible modifications, energy consumption decreased to a level just slightly higher than the P2P-disabled official app on both Wifi and 3G.

A survey by Hoque et al. [9] showed numerous approaches at optimizing energy consumption of multimedia streaming on Wifi, 3G and LTE networks found in the literature, categorized

[2]Tribler [1] is an existing P2P application. We just modified it to support boosting.

[3]http://www.computerworld.com/s/article/9243331/ (Nov. 2013)

by layer: physical, link, application, and cross layer. Hoque and his colleagues went on to propose their own approach using crowd-sourced viewing statistics and evaluated it on 3G and LTE networks [10]. On these networks, there is a trade-off between (1) pre-fetching large chunks at the risk of downloading useless data if the user stops watching, (2) and fetching small chunks, at the risk of increasing total energy consumption because wireless interfaces stay powered for several seconds after each data transfer. Although these studies focused on client-server streaming, they provide insights that are also applicable to P2P streaming on mobile devices.

C. NAT Gateways

Network address translator (NAT) gateways are used to allow multiple devices to access the Internet using a single IP address. Practically, every Wifi-enabled router (including ADSL, and cable modems provided by Internet service providers) features NAT functionality with few exceptions (e.g., KTH's Wifi network provides public IP addresses). That is, most Wifi connections go through a NAT gateway.

Hätönen et al [11] studied 34 different home gateway models and observed noticeable differences among NAT implementations, reporting that no gateway used the parameter values recommended by the IETF standard for NAT gateways [12].

Mobile networks also use NAT mechanisms, mainly to multiplex an ever-scarcer number of IP addresses. Mäkinene and Nurminen [13] characterized the NAT policies of six major mobile operators, observing that existing TCP NAT traversal techniques work in the majority of these networks. Wang et al. [14] developed a mobile app which allowed them to collect data regarding NAT and firewall policies deployed in over 100 mobile ISPs. Less than a quarter of the these mobile ISPs provided public IP addresses.

NAT gateways supports relatively well client-server applications where clients initiate connections and the server has a public IP address. When a connection is established by a host behind a NAT gateway (client), the gateway will forward packets between client and server, for the most part.

On P2P networks, peers should ideally be able to establish connections to each other because each peer is able to both consume and offer services. On the Internet, however, NAT gateways severely restrict connectivity, making it hard to establish a connection to a peer behind a NAT gateway.

In general, NAT gateways hinder P2P performance [15]. When the only peer with a copy of the content is behind a NAT gateway, content distribution might never start. This paper addresses the particular problem of a piece of content's original source being behind a NAT gateway.

There are many NAT traversal mechanisms to allow hosts behind NAT gateways to establish direct connections to each other. For instance, NatCraker [16], Usurp [17]. Halkes and Pouwelse's UDP NAT puncturing [18] provide an overview of UDP NAT traversal techniques used on the Internet. While our current prototype does not use any of these mechanisms, we are considering integrating some of these mechanisms in future versions.

D. PPSP and Libswift

The Peer-to-Peer Streaming Protocol (PPSP) suite [19] is currently in the process of being standardized at the Internet Engineering Task Force (IETF)[4]. *Libswift*[5] is the reference implementation of PPSP's transport protocol (previously known as *Swift* [20], [21]) and it is available under an open-source license.

PPSP inherits many of its properties from BitTorrent [6], while introducing several improvements. Unlike BitTorrent, PPSP was explicitly designed for multimedia streaming, both on-demand and live; although it can also be used as generic multi-party transfer protocol.

Since PPSP is based on UDP, it is easier to traverse NAT gateways and achieve lower delays. PPSP uses a hashing scheme (Merkle hashes) that greatly improves the dissemination of integrity metadata, reducing user-perceived start-up delay compared to BitTorrent, specially when considering large amounts of data such as a high-definition feature film. While the few hashes needed by a PPSP peer to start checking integrity are likely to fit in a single UDP, a BitTorrent peer must obtain every single hash (several kilobytes packed in a *.torrent file*) before being able to perform any integrity check.

PPSP has been deployed on desktop computers, mainly through its integration on Tribler [1] and a browser plug-in project in collaboration with Wikipedia [22]. Preliminary experiments of *libswift* on Android showed that it was feasible for a modern smartphone to run *libswift* [23].

PPSP has also been proposed as a P2P-based implementation of information-centric networks (ICN). PPSP's Merkle hashes provide naming properties very similar to those proposed in other ICN systems, making PPSP a good candidate for piece-meal adoption on the current IP-based network infrastructure, unlike other ICN proposals that require a clean-slate redesign of the networking infrastructure [20], [21].

III. Roles and Interactions

Before we describe the different parts of the system, we define the different roles in the system and an example to illustrate how the system works and how the different roles interact with each other.

A. Roles

We define roles broadly because our goal is to design a flexible system that is able to support different applications: from best-effort community services to commercial services with strict quality of experience requirements.

- **source**
  User has a piece of content that does not exist in the P2P network yet. Sources can upload content to peers, leechers, and boosters. A source running on a mobile device is referred as **mobile source**.

[4]PPSP Working Group: https://datatracker.ietf.org/wg/ppsp/ (Nov. 2013)
[5]http://libswift.org/ (Nov 2013)

- **peer**
  User downloads and watches video, uploading it to others as a side-effect (as it happens in P2P-based file-sharing systems). Although some mobile devices would be able to function as peers, we conservatively assume that all mobile devices are leechers. Thus, we will use the term **desktop peer** in this paper.
- **leecher**
  Typical operation of mobile devices (**mobile leecher**). Leechers can download and playback the content, but unlike peers, leechers do not upload. We do not consider leechers as free-riders and the system should provide them whenever possible.
- **booster**
  Peer that is configured to automatically redistribute (*seed*) content according to some user-controlled policy (e.g., subscription).
- **meta-booster**
  User who posts content metadata to a subscription service. Boosters subscribed to this meta-booster will automatically start boosting the content, according to their policies.
- **sharer**
  User who helps spreading the content in the same way it is done in social media platforms. That is, sharing pointers (e.g., links) to the data instead of distributing the data themselves.

B. Community-Supported Video Streaming: an Example

The NGO "Bits International" encourages its members to record videos with their smartphones to document their actions and raise awareness. BI members have published their videos on web-based video sharing platforms before but a recent video upload was too controversial and was taken down alleging it was "inappropriate", despite the fact it was a clear exercise of freedom of speech and nobody questioned its legality.

Our example is fictional but takedowns do happen in actual platforms, mainly related to controversial topics[6] and dubious copyright takedowns[7]. In the absence of a court order, the platform owner must decide whether to allow a particular video to be distributed or not.

BI has now decided to use Tribler Mobile to distribute their videos and asks its members to collaborate by sharing and boosting these videos. Now, as long as users donate enough resources in the form of bandwidth and storage (via **boosters**), content can be distributed to a large audience.

Adam, one BI activist, records a new video on his smartphone and uses Tribler Mobile to upload it. The smartphone (**mobile source**) is ready to send the data to others. The app also generates metadata that Adam posts (**meta-booster**) on a subscription service (red dashed arrow from Adam's mobile source in Figure 1). As soon as the metadata is posted, all **boosters** subscribed to Adam's content will download the data from the mobile source and each other (solid black arrows among Adam's mobile source and Adam's boosters). Once a complete copy of the data has been uploaded, Adam can stop uploading and exit the app, thus saving battery. Notice that we have not specified how subscription works nor how mobile source and boosters find each other; we will cover those mechanisms in detail in Sections IV-A and IV-B, respectively.

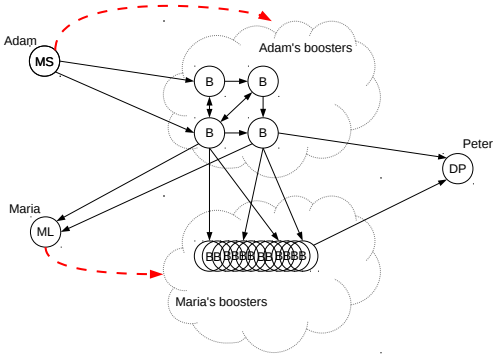

Fig. 1. Interaction among a mobile source (MS), boosters (B), a mobile leecher (ML), and a desktop peer (DP). Dashed red arrows represent the flow of meta-data from publisher to subscribers. Solid black arrows represent data flows.

Adam also shares a link (**sharer**) on social networks to notify *followers* and *friends* that there is a new video available for streaming. Maria, one of Adam's followers, opens the link on her smartphone (**mobile leecher**), streaming the video from boosters (notice that, in this case, the mobile source is already off-line). Maria likes the video and posts the link on the subscription service (**meta-booster**). This action adds much distribution capacity since hundreds of boosters (including her own computer at home) are subscribed to Maria. Maria is happy to contribute in ways that do not involve draining her battery and mobile data traffic.

Peter opens the link Adam shared on his computer and streams the video from boosters, his computer not only downloads data but it also uploads it to others (**desktop peer**).

In this example, as in our prototype, boosters boost all content they are subscribed to. This is not a limitation of the system in itself but just a simplification to accelerate prototype implementation and deployment. More advanced boosters would decide whether to boost a piece of content depending on different factors.

We define the term booster broadly to allow for a broad spectrum of boosters that may be combined with incentive mechanisms, both centralized (e.g., private BitTorrent trackers [24], [25], [26]) and distributed (e.g., BarterCast [27], [28]

[6]http://news.cnet.com/8301-1023_3-57513354-93/no-easy-outs-for-youtube-in-islam-video-controversy/ (Nov. 2013)

[7]https://www.eff.org/press/releases/lawrence-lessig-strikes-back-against-bogus-copyright-takedown (Nov. 2013)

and Dispersy [29]). In this paper, however, we will mainly focus on *altruistic boosters*.

IV. Infrastructure Services

We use PPSP (introduced in Section II-D) as transport protocol for peers to transfer data to each other but other infrastructure services are needed to coordinate peers.

This section describes each of the main infrastructure services supporting the system. For each one, we first expose general requirements and challenges, proposing generic mechanisms which are open to different implementations, according to the specific needs and resources. Then, we provide a description of our prototype's implementation details and discuss their properties and limitations. In some cases, we discuss alternative implementations that may be used in different contexts.

A. Peer Discovery

In the example in Section III-B (Figure 1), peers connect and transfer data to each other. How does a peer find one or more peers where the requested data is stored and ready to be transferred? A *peer discovery mechanism* keeps track of peers sharing a piece of content and peers can request a list of peers, given a content identifier (identifiers are a hash string derived from the data itself and can be used for integrity protection).

Popular P2P networks use two complementary kinds of peer discovery mechanisms: tracker and gossip-like. Some systems (e.g., Spotify) use both of them [2]. In this paper, we focus on tracker-like peer discovery.

A tracker is a centralized registry where all peers sharing a piece of content are registered. For each request, the tracker returns a short list of peers (if there is any). Trackers started as centralized services running on a single machine and have evolved to become decentralized services. For instance, nowadays BitTorrent can use both centralized trackers, where the service is run on a single (or few) server(s), and decentralized DHT-based trackers, where the service is run by millions of DHT-enabled BitTorrent peers.

1) Challenges: One of the side-effects of IPv4's address exhaustion is the massive deployment of NAT gateways. NAT gateways are commonly used where several devices need Internet connection but only one public IP address is available. Nowadays, most Wifi access points are also NAT gateways and much of the mobile operators do not provide a public IP address, but a private one behind a large NAT gateway [13], [14]. Therefore, mobile devices connected through Wifi or mobile networks are very likely to be *NATed*.

NAT gateways are a great challenge for P2P networks because NATs restrict connectivity. In general, it is difficult to establish a connection to a *NATed peer* and even harder if both are NATed. Several techniques have been proposed and deployed but NATs remain a main challenge for P2P-based systems.

This paper focuses on the particular challenge of uploading new content from a mobile source. Since it is the only source available, we must guarantee that a connection can be established between the mobile source and, at least, a peer (a booster in our case).

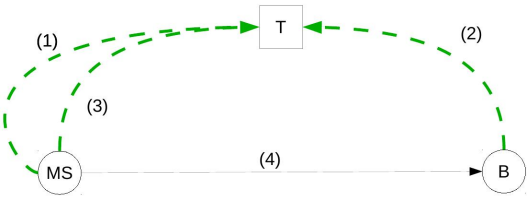

Fig. 2. Reverse peer discovery. **(1)** Mobile source (MS) periodically requests peers from the tracker (T) but it does not announce itself, T returns an empty list. **(2)** a booster (B) requests peers from T and it announces itself, T returns an empty list. **(3)** MS requests peers, T returns a list containing B. **(4)** MS establishes a connection to B, data transfer begins.

Popular P2P systems like BitTorrent are pull-based systems. In BitTorrent, a peer offering new content (a smartphone with the original copy, in this case) would register themselves in the peer discovery service (tracker) and wait for other peers to establish connections. Unfortunately, that smartphone is likely to be NATed, making it very hard to establish connections from other peers to the smartphone.

Mobile devices with an open connection (i.e., not NATed) are challenging in a different way. Normally, once a peer is registered in a tracker, this peer cannot unregister itself. Instead, the tracking mechanism only *unregisters* peers if they have not renewed their registration within a period of time of (commonly several minutes). Thus, if a mobile peer registers itself and it happens to be reachable, it runs the risk of receiving connections attempts over a long time period, which brings the radio to a high-power state, thus consuming extra energy. These connection attempts will reach the device even after the P2P app no longer runs on the device.

Thus, we need to make sure that he mobile peer does not register itself in the tracker to limit energy consumption.

2) Booster Discovery: We have reversed peer discovery so that the mobile source establishes connections to other peers, to increase the success rate of mobile sources uploading new content to the P2P network and reduce energy consumption.

We present now a general description of the mechanism. Then, we provide implementation details regarding our prototype implementation.

A mobile source uploading new content to the P2P network posts the content's identifier on the subscription service (Section IV-B) and requests a list of peers from the tracker in few-seconds intervals, as shown in Figure 2.

As soon as a booster is notified of a new piece of content, the booster requests peers and registers itself in the tracker. In the next periodic tracker request, the mobile source will receive the booster's address (IP address and UDP port) and establish a connection to the booster. If the booster were not reachable, the mobile source keeps querying the tracker for more peers until it establishes connections and transfers a full copy of the content. At that point, the mobile app can be

stopped. Boosters will continue distribution to other boosters, peers, and mobile leechers.

Ensuring successful data upload from a mobile source is relatively simple and cheap. The user himself (or a trusted party) can set up a non-NATed booster at home or on a rented server and configure it to boost his own content.

It is worth emphasizing that this simple reverse mechanism allows us to build a fully decentralized streaming service where mobile devices can upload content even in the presence of NAT gateways. We plan to include other NAT traversal mechanisms in the future.

3) Prototype Implementation: In our prototype, we aim for a fully-decentralized system and thus do not rely on a centralized tracker. Instead, we use a DHT-based mechanism: BitTorrent's Mainline DHT (MDHT).

In particular, we use Pymdht, a MDHT implementation in Python designed to collect experimental data and to be easy to modify. We have used this implementation before [30], [31], and it is publicly available as open-source software. In this case, we modify Pymdht to support the reverse lookup mechanism described above.

On our modified Tribler desktop application, boosters that lookup a content identifier and find no peers, announce themselves to the DHT-based tracker even though they have no content to offer. This is done by sending an `announce_peer` request to the appropriate nodes in the DHT overlay.

On the Tribler Mobile app, there are two different configurations: mobile source and mobile leecher.

The mobile source sends `get_peers` requests to the DHT periodically (20 seconds in the current implementation) but it does not announce itself. As soon as the DHT returns a lists of boosters, the mobile source establishes connections to a subset of them and uploads data. The app user is shown a status screen and he should wait until a full copy of the content has been transferred before closing the app. Once the app is closed, the app will not send/receive traffic, limiting energy consumption.

The mobile leecher performs a single DHT lookup, sending `get_peers` requests to DHT nodes, and as the mobile source, it does not announce itself. As soon as the DHT returns peers' addresses, the mobile leecher establishes connections and downloads data from them, starting playing back as soon as the playback buffer is sufficiently populated.

There are two other major differences between the mobile and the desktop implementations. First, while the desktop implementation is basically vanilla Pymdht, the mobile implementation does not perform any DHT routing and maintenance tasks, ignoring all requests from other nodes. There are two reasons for this: (1) NATed DHT nodes cannot properly route DHT lookups, harming lookup performance for other nodes if they try [30], and (2) to avoid future requests from other peers that would power up the radio interface even after the app has been closed.

The second difference is that we translated Pymdht from Python to Java due to the current poor support for Python libraries on Android. Nevertheless, the translation was done with great care to make both Java and Python implementations equivalent.

B. Subscription Service

Boosters require a subscription service to receive notifications of newly published content metadata. Once a booster receives a notification, it will decide whether to boost (i.e., fetch and redistribute the data), according to its configuration. The functionality needed from the subscription service will depend on the boosting incentive mechanisms deployed in the system.

Our goal is to design a flexible system that supports a wide range of boosting incentives. On the other hand, our prototype provides a concrete implementation of a simple mechanism as a proof of concept.

Existing P2P-based file-sharing systems have different incentive mechanisms. BitTorrent, for instance, only has incentives for peers currently downloading data from other peers (tit-for-tat [32] and super-seeding [33]). There are only indirect incentives for *seeding* (uploading data to others after the peer has completely downloaded a piece of content). If there are enough users seeding, it is likely that the peer that seeded one piece of content at some point in time will download a different piece of content from another altruistic peer in the future. This may be viewed as a kind of indirect reciprocity.

Private BitTorrent trackers introduce a direct incentive mechanism for seeding. A private tracker keeps track of each peer's share ratio (bytes uploaded divided by bytes downloaded). Typically, the tracker enforces a minimum ratio by refusing to return a list of peers to those peers whose ratio is lower than the minimum [25], [26], [24].

BarterCast [27] allows peers to trade bandwidth among each other in a decentralized manner, converting bandwidth into currency. Our long-term goal is to integrate a similar mechanism to support fully-decentralized boosting incentives.

1) Prototype Implementation: Our prototype uses Twitter as subscription service. We first describe how it is used with an example, and then discuss its advantages and limitations.

When Bob wishes to contribute to the distribution of content published by Alice, Bob simply adds Alice's Twitter handle (`@alice`) to his booster's configuration. Then, Bob's booster will periodically monitor `@alice` for new *tweets* containing content metadata in the form of a PPSP URI schema (i.e., ppsp://f37...f1). Shortly after Alice *tweets* the PPSP URI, Bob's booster is notified of the new content and it starts downloading and redistributing the content whose identifier is f37...f1. It is worth mentioning here that the identifier is derived through cryptographic hash derived from the data, providing data integrity protection.

As a side-effect, Alice is also *sharing* a PPSP URI. That means that Alice's followers can click on the URI and start streaming video to their devices (either desktop or mobile) from boosters. If any of these followers re-posts (*retweet*) the URI, the boosters subscribed to that follower will also start boosting, adding distribution capacity.

Fig. 3. Posting of a PPSP.me link on Twitter.

We could have used RSS subscription, which most BitTorrent clients already support. Twitter, however, has millions of interconnected users [34] and offers additional user-friendly features (retweet in particular) that produce powerful network effects [35]. In our case, they enable fast and wide dissemination of content metadata.

Other advantages of using Twitter as boosting subscription mechanism are: (1) it couples content to a person or organization, making it relatively simple to decide where to donate resources (via boosting), and (2) a single system handles meta-boosting and sharing.

The obvious limitation of using Twitter is having a single point of failure and the risk that Twitter would not accept PPSP URIs for commercial, legal, or any other reason. That is why our long-term goal is to switch to a fully-distributed pub-sub service.

C. PPSP.me link translator

This part is technically not part of the system, but just a convenient utility to bootstrap deployment. We include it in this paper because we consider incremental deployment a critical part of a successful system.

Tribler Mobile uses its own URI schema: ppsp://hash. Unlike HTTP URLs (e.g., http://kth.se/en), PPSP URIs do not contain any information regarding the content's location. In fact, PPSP could be considered information-centric, as discussed in Section II-D. Instead, data's location is discovered via a peer discovery service, as we discussed in Section IV-A.

Unfortunately, browsers and mobile apps do not implement any handle for PPSP URIs yet, showing an error to the user who tries to open the *PPSP link*.

To bootstrap deployment, we have implemented a simple link translator which leads users to useful information about Tribler Mobile and how to playback content referred by a PPSP link.

Thus, when Alice records a video on her smartphone and chooses to publish it via Tribler Mobile, our software will generate an HTTP URL in the form of http://ppsp.me/hash for Alice to *tweet* it (see Figure 3 for an actual example).

Then, Alice's followers who click on the ppsp.me link, they will land on a simple web page with instructions to install Tribler Mobile. Once Tribler Mobile is installed, the user is asked whether he wants to always open ppsp.me links using Tribler Mobile instead of a browser.

While the ppsp.me web service is a single point of failure, it is not a critical service, just a convenient way of introducing new users to Tribler Mobile. Users who already have Tribler Mobile installed do not contact ppsp.me servers at all, and directly proceed to find peers and stream content from them. Even users who have not installed Tribler Mobile yet can find information on the web about how to install the app, should ppsp.me be temporally (or even permanently) unavailable.

Notice that ppsp.me links have no effect on boosters, since boosters can parse both PPSP URIs and ppsp.me links, and treat them equally.

V. Conclusion

We have presented a P2P video streaming platform where mobile devices are well integrated, considering their limitations: battery, mobile data caps, and limited connectivity due to NAT gateways.

In particular, we are the first to propose, implement, and deploy a complete system where mobile devices can publish content directly into the P2P network and let other peers (boosters) distribute the content to a potentially large audience (including other mobile devices).

Our contributions include a reverse peer discovery mechanism that addresses two important issues. First, it allows mobile devices to traverse NAT gateways —these gateways are widely-deployed on Wifi and mobile networks. Second, it saves battery because the mobile device is never registered on the peer discovery service, thus other peers will never contact the mobile device —the device's radio must switch to high-power state when receiving packets.

Our prototype has been built incrementally. Whenever possible we have used existing proven-to-work components, implementing backwards-compatible modifications to adapt each component to our requirements. For instance, our reverse peer discovery mechanism is backwards-compatible with BitTorrent's Mainline DHT, allowing us to leverage this multi-million DHT overlay.

Our long-term goal is to evolve this prototype into a fully-decentralized system without any single point of failure (our current prototype relies on Twitter). Furthermore, we plan to integrate more sophisticated NAT traversal mechanisms, and incentive mechanisms to create a marketplace for publishers and boosters.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the Seventh Framework Programme (QLectives), SSF (E2E-Clouds), and EIT ICTLabs.

The authors would like to thank Elric Milton, Niels Zeilemaker, Mihai Capotă for their help setting up the infrastructure.

REFERENCES

[1] 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.

[2] G. Kreitz and F. Niemela, "Spotify - Large Scale, Low Latency, P2P Music-on-Demand Streaming," in *P2P'10*, 2010.

[3] B. Bakos, G. Csúcs, L. Farkas, and J. K. Nurminen, "Peer-to-peer protocol evaluation in topologies resembling wireless networks. an experiment with gnutella query engine."

[4] B. Bakos, L. Farkas, and J. K. Nurminen, "P2p applications on smart phones using cellular communications," in *Wireless Communications and Networking Conference, 2006. WCNC 2006. IEEE*, vol. 4. IEEE, 2006, pp. 2222–2228.

[5] J. Nurminen and J. Noyranen, "Energy-consumption in mobile peer-to-peer - quantitative results from file sharing," in *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, 2008, pp. 729–733.

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

[7] I. Kelényi and J. K. Nurminen, "Energy aspects of peer cooperation measurements with a mobile dht system," in *Communications Workshops, 2008. ICC Workshops' 08. IEEE International Conference on*. IEEE, 2008, pp. 164–168.

[8] R. Jimenez, G. Kreitz, B. Knutsson, M. Isaksson, and S. Haridi, "Integrating Smartphones in Spotify's Peer-Assisted Music Streaming Service."

[9] M. Hoque, M. Siekkinen, and J. Nurminen, "Energy efficient multimedia streaming to mobile devices — a survey," vol. PP, no. 99, 2012, pp. 1–19.

[10] M. A. Hoque, M. Siekkinen, and J. K. Nurminen, "Using crowd-sourced viewing statistics to save energy in wireless video streaming," in *Proceedings of the 19th annual international conference on Mobile computing and networking*, ser. MobiCom '13. New York, NY, USA: ACM, 2013, pp. 377–388.

[11] S. Hätönen, A. Nyrhinen, L. Eggert, S. Strowes, P. Sarolahti, and M. Kojo, "An experimental study of home gateway characteristics," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, ser. IMC '10. New York, NY, USA: ACM, 2010, pp. 260–266. [Online]. Available: http://doi.acm.org/10.1145/1879141.1879174

[12] R. Braden, *RFC 1122 Requirements for Internet Hosts - Communication Layers*, Internet Engineering Task Force, Oct. 1989. [Online]. Available: http://tools.ietf.org/html/rfc1122

[13] L. Makinen and J. K. Nurminen, "Measurements on the feasibility of tcp nat traversal in cellular networks," in *Next Generation Internet Networks, 2008. NGI 2008*. IEEE, 2008, pp. 261–267.

[14] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang, "An untold story of middleboxes in cellular networks," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 374–385, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2043164.2018479

[15] B. Ford, "Peer-to-peer communication across network address translators," in *In USENIX Annual Technical Conference*, 2005, pp. 179–192.

[16] R. Roverso, S. El-Ansary, and S. Haridi, "Natcracker: Nat combinations matter," in *Computer Communications and Networks, 2009. ICCCN 2009. Proceedings of 18th Internatonal Conference on*, 2009, pp. 1–7.

[17] S. Niazi and J. Dowling, "Usurp: Distributed nat traversal for overlay networks," in *Distributed Applications and Interoperable Systems*, ser. Lecture Notes in Computer Science, P. Felber and R. Rouvoy, Eds. Springer Berlin Heidelberg, 2011, vol. 6723, pp. 29–42.

[18] G. Halkes and J. Pouwelse, "Udp nat and firewall puncturing in the wild," in *NETWORKING 2011*, ser. Lecture Notes in Computer Science, J. Domingo-Pascual, P. Manzoni, S. Palazzo, A. Pont, and C. Scoglio, Eds. Springer Berlin Heidelberg, 2011, vol. 6641, pp. 1–12.

[19] A. Bakker, P. R., and V. Grishchenko, "Peer-to-Peer Streaming Peer Protocol (PPSPP)," 2013.

[20] F. Osmani, V. Grishchenko, R. Jimenez, and B. Knutsson, "Swift: the missing link between peer-to-peer and information-centric networks," in *Proceedings of the First Workshop on P2P and Dependability*, ser. P2P-Dep '12. New York, NY, USA: ACM, 2012, pp. 4:1–4:6. [Online]. Available: http://doi.acm.org/10.1145/2212346.2212350

[21] 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, Tech. Rep., 2011.

[22] 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.

[23] R. Petrocco, J. Pouwelse, and D. Epema, "Performance analysis of the libswift p2p streaming protocol," in *12-th IEEE International Conference on Peer-to-Peer Computing (P2P12)*. IEEE, 2012. [Online]. Available: http://dx.doi.org/10.1109/P2P.2012.6335790

[24] J. J.-D. Mol, J. A. Pouwelse, D. H. Epema, and H. J. Sips, "Free-riding, fairness, and firewalls in p2p file-sharing," in *Peer-to-Peer Computing, 2008. P2P'08. Eighth International Conference on*. IEEE, 2008, pp. 301–310.

[25] M. Meulpolder, L. D'Acunto, M. Capotă, M. Wojciechowski, J. A. Pouwelse, D. H. J. Epema, and H. J. Sips, "Public and private bittorrent communities: a measurement study," in *Proceedings of the 9th international conference on Peer-to-peer systems*, ser. IPTPS'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863145.1863155

[26] Z. Liu, P. Dhungel, D. Wu, C. Zhang, and K. Ross, "Understanding and improving ratio incentives in private communities," in *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, 2010, pp. 610–621.

[27] M. Meulpolder, J. Pouwelse, D. H. J. Epema, and H. Sips, "Bartercast: A practical approach to prevent lazy freeriding in p2p networks," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, 2009, pp. 1–8.

[28] R. Delaviz, N. Andrade, and J. A. Pouwelse, "Improving accuracy and coverage in an internet-deployed reputation mechanism," in *Peer-to-Peer Computing'10*, 2010, pp. 1–9.

[29] N. Zeilemaker and J. Pouwelse, "Open source column: Tribler: P2p search, share and stream," *SIGMultimedia Rec.*, vol. 4, no. 1, pp. 20–24, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2206765.2206767

[30] 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.

[31] ——, "Sub-second lookups on a large-scale kademlia-based overlay," in *11th International Conference on Peer-to-Peer Computing 2011*, Kyoto, Japan, 8 2011.

[32] B. Cohen, "Incentives to Build Robustness in BitTorrent," in *Proceedings 1st Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, Jun. 2003, http://bittorrent.com/bittorrentecon.pdf.

[33] Z. Chen, Y. Chen, C. Lin, V. Nivargi, and P. Cao, "Experimental analysis of super-seeding in bittorrent," in *Communications, 2008. ICC '08. IEEE International Conference on*, 2008, pp. 65–69.

[34] M. Gabielkov and A. Legout, "The complete picture of the twitter social graph," in *Proceedings of the 2012 ACM conference on CoNEXT student workshop*. ACM, 2012, pp. 19–20.

[35] D. Boyd, S. Golder, and G. Lotan, "Tweet, tweet, retweet: Conversational aspects of retweeting on twitter," in *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*, 2010, pp. 1–10.

Reader's Notes