SILVERSTREAM

A FULLY DECENTRALISED MUSIC STREAMING PLATFORM

ALEXANDER MUNCH-HANSEN, 201505956

CASPER VESTERGAARD KRISTENSEN, 201509411

BUILDING THE INTERNET OF THINGS WITH P2P AND CLOUD COMPUTING

December 2018

Advisor: Niels Olof Bouvin

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

SILVERSTREAM

ALEXANDER MUNCH-HANSEN
CASPER VESTERGAARD KRISTENSEN

A fully decentralised music streaming platform

Building the Internet of Things with P2P and Cloud Computing

Department of Computer Science
Science & Technology
Aarhus University

December 2018

ABSTRACT

Peer to peer technologies effectively allow for transforming previously centralised technologies to work in a completely decentralised way, by allowing the spreading of the work-load to multiple peers. In this project we develop a scalable and reliable music streaming platform that is completely decentralised. This platform utilises the `Mainline-DHT` to discover music which can then be played by the end-users of the platform. We present results of trials using varying algorithms for discovering songs and ways of acting in the `Mainline-DHT` network. The results show a clear difference, when we alter these things.

CONTENTS

1 INTRODUCTION

The music streaming business is a highly competitive and centralised one, and it has spawned several big companies such as *Spotify* and *Deezer*. Because of this, all major services within this subject are highly centralised and closed source. As such, the companies are in full control of their platforms, leading the users of the systems to having to rely on the company keeping their service and servers alive. Additionally, the users have to rely on the company deeming it worthy to buy the rights to the music they want to listen to. As such, the users have negligible control over what music is available to them.

In this project, we design and implement a fully decentralised music streaming platform, `Silverstream`, as a response to the growing centralisation of the music streaming industry. Being a peer-to-peer system, it relies heavily on users joining together and allowing others to leech from them. `Silverstream` uses the `DHT` of the `BitTorrent` network, known as the "`Mainline-DHT` ", as a way of discovering music and to increase the amount of music available in the system. This is accomplished by inserting a large number of `DHT` nodes into the `Mainline-DHT`, which allows for passively listening to the traffic within the network. By processing this information, the available music can be discovered, and in the end the user is capable of searching and playing back music directly from the `BitTorrent` network.

We experiment with different algorithms for inserting crawler nodes into the `BitTorrent` network, as well as differ the behaviour of the nodes themselves, in an attempt to find a golden ratio between the amount of bandwidth the nodes use versus the information gained. This requires careful testing, as the `Mainline-DHT` is a living organism with millions of nodes worldwide leaving and joining at all times of the day.

All parts of the system are custom built for this project: This includes a framework built on top of the `LibTorrent` framework[1], which is used by many well-known torrent clients such as *qBitTorrent* and *Deluge*, a framework for playing music using *mpv* and our own `Kademlia` implementation. While these frameworks are very feature-packed, it is important to emphasise that we use them only sparingly; in particular, our use is limited to things that are irrelevant to this course, such as the details of playing different audio formats or the `BitTorrent` metadata-exchange protocol between two peers. We have implemented a fully-functioning `Kademlia` DHT node conforming to the requirements of the `BitTorrent` protocol, allowing us exactly the

1 https://libtorrent.org

behaviour we desired and flexibility during experimentation – this was particularly important to us, as we did not want to disrupt the traffic of the live `Mainline-DHT`.

In recent years, there has been a major push for applications to include more decentralised components, as opposed to the completely centralised *Server-Client* architecture; this tends to allow for a more robust application, as it moves away from the single point of failure of the server. Additionally, the peer-to-peer architecture allows companies cost-savings by moving some of the work-load on to the users of the system, potentially allowing for cheaper solutions. However, while great in theory, peer-to-peer systems also tend to be more difficult to keep secure and running smoothly, since it depends on users participating honestly with a good internet connection. A music-streaming platform additionally requires that enough peers are distributing the music for the system to be fast enough to stream songs. Because of this, most torrent system works on a *tit-for-tat* policy, such that all users have to cooperate, share and in general participate nicely, for them to not get booted from the network.

2 RELATED WORK

As mentioned in the introduction, not many decentralised music streaming projects have been proposed before. As such, we focus on related work in regards to the individual parts of the system.

2.1 PRELIMINARIES

We will throughout the report assume that the reader is somewhat familiar with basic concepts within networking and communication. Because we consider the details of DHT's and the Mainline-DHT in specific as well as Kademlia as being slightly more obscure, we here describe them in necessary detail.

2.1.1 *DHT*

Distributed hash tables (DHTs) are a class of a decentralised distributed systems that provides a look-up service similar to a hash table. It contains (key, value)-pairs and any participating node can efficiently retrieve the value associated with a given key. Keys are unique identifiers, which map to particular values, which in turn can be arbitrary data. Responsibility for maintaining this mapping from keys to values is distributed among the nodes in such a way that a change in the set of participants causes a minimal amount of disruption. This allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures. This makes DHTs an ideal way of storing information in unstable *peer-to-peer* networks with high churn.

2.1.2 *mainline-DHT*

Mainline-DHT is the name given to the Kademlia-based DHT used by BitTorrent clients to find peers via the BitTorrent protocol. The idea of utilising a DHT for distributed tracking was first implemented in *Azureus*, which is now known as VUZE, from which it gained significant popularity. Shortly after, BitTorrent Inc. released their own DHT into their client, called Mainline-DHT and thus popularised the use of distributed tracking in the BitTorrent Protocol.

2.1.3 *Kademlia*

Kademlia is a distributed hash table for decentralised peer-to-peer. It specifies the structure of the network and the exchange of information through node look-ups and as such specifies a structured peer-to-peer network.

Kademlia does not specify a way of picking node-id, this allows for two nodes which might be geographically in two very different locations, to be relatively close within Kademlia. This specification is being questioned by a *BitTorrent Enhancement Proposal* (BEP), but as of writing this, it is still allowed to determine your own node-id. Kademlia routing tables consist of a list for each bit of the node ID, so each node will keep 160 lists, if the id-space is 160 bits. Every list corresponds to a specific distance from the node and this distance is computed by calculating the XOR between two node-ids. Nodes that can go in the nth list must have a differing nth bit from the nodes node-id. In Kademlia, these lists are referred to as k-buckets. K is simply a system wide number, which can be set as the user sees fit. Every k-bucket is a list having up to k entries inside.

2.2 RELATED WORK

In general, much work and effort have gone into the crawling and monitoring of users and torrents in torrent DHTs.

One such system was built by Scott Wolchok and J. Alex Halderman[7]. They created a crawler which only supported the VUZE-DHT and was not capable of running on Mainline-DHT without modifying it first. This VUZE-DHT crawler was called *ClearView* and its main purpose was to be able to crawl and build a database over the torrents existing in the VUZE-DHT. They used a technique to gather data by the means of what is essentially a sybil attack. A sybil attack is when you insert a lot of nodes into a network and you start to control what goes on, as you can use these nodes to route poorly and spread misinformation. *ClearView* however used these nodes to passively listen and whenever they were asked to replicate data, they would log this. Furthermore, the nodes jumped around the address space in an attempt to gather more data while re-using nodes and thus saving compute power. This Crawler was then utilised in another project called *SuperSeed*. *SuperSeed* uses *ClearView* to crawl the VUZE-DHT and make the user of the system, capable of scraping torrents and gathering a database over what the crawler finds. As *SuperSeed* uses *ClearView*, this system can only be used for the VUZE-DHT. There is one main difference as to why this is; The Mainline-DHT doesn't support replication, which is what *ClearView* extensively used. As such, *SuperSeed* can't be ported to work as efficiently on the Mainline-DHT without fundamentally changing the system.

Memon et al. developed a system, *Montra*, for monitoring the traffic within the DHT, where the DHT monitored is unspecified[5]. As the system attempts to monitor what gets shared and such, it inserts peers into the DHT and passively watches. The purpose of *Montra* was to question how many peers to insert into the DHT to maximise the information gathered. If too many peers are inserted into a DHT, this might disrupt the traffic as the newly inserted nodes won't add any information. While too many nodes might be an issue, so can too few nodes be. If too few are inserted into the DHT, you might not be able to get a great enough understanding of the traffic circulating the DHT. Therefore, Memon et al. sought to find if there was a diminishing return on the amount of nodes to insert. Memon et al. managed to concurrently run 32.000 nodes without disrupting traffic and concluded this was optimal, as they managed to capture 90% of the traffic.

Scott A. Crosby and Dan S. Wallach have investigated both how well the look-ups in the Mainline-DHT works and how long they on average take[1]. They conclude that roughly 20% of all look-ups would end up in dead-ends and that a lookup on average would take around a minute. However, Crosby and Wallach performed these tests on a version of the Mainline-DHT which had several performance impacting bugs at the time of their testings, which they would fix before doing their testings, as such, the Mainline-DHT was likely slower at the time.

Cubit is a system created with the purpose of supporting full text search and is implemented by Wong et al. This system uses the *Levenshtein* distance, an *edit-distance*, to determine how close two keywords are to each other. A node in the Cubit network has a specific keyword taken from the list of files in the system. Each node keeps track of something called *multi-resolution rings*, which each node keeps updated with other nodes whose keyword is a specific Levenshtein distance away from its own keyword. Searching is done by giving a keyword to a node and letting this node calculate the edit distance, thus determining where to send the request to. As the Levenshtein distance isn't defined for phrases, they establish a term called the Additive Minimum Edit-Distance, which is essentially the Levenshtein distance calculated over each keyword of the phrase. They conclude their system drastically reduces message overhead, as they avoid using flooding for searching and they use simple gossipping when a node joins or when a node is noticed to be dead.

While the aforementioned systems have remained mostly in the academic world, other have gained tremendous popularity among the general public. These systems include software like Napster for music streaming, Popcorn Time for movie streaming, and the BitTorrent protocol, used for general file sharing. Common for the two former systems is the fact that the file directory relies on a centralised component, while only the bandwidth-expensive operation of actu-

ally sharing the files are built on a peer-to-peer architecture. This design decision makes it relatively easy for the authorities to shut down the operation of the networks by targeting the centralised component, as was the case with Napster[2].

Unlike Napster, the `BitTorrent` protocol can operate completely without any centralised directory service[4]. This, however, requires the user to gain knowledge of the file-ID they wish to download out-of-band. In the context of `BitTorrent`, most users do this through a centralised directory called a `BitTorrent` tracker. The Popcorn Time software uses a set of these trackers to search, discover, and play movies over the `BitTorrent` protocol[3].

The centralisation of a component like the `BitTorrent` tracker can have many benefits, and in most cases helps increase the performance and usability of the system, but it can also decrease the system's resistance to censorship. A well known example of this centralisation being a problem was with the blocking of the `BitTorrent` tracker The Pirate Bay in many countries around the world, which sought to make it more difficult for the users to discover files[6].

3

ANALYSIS

While most, if not all, of the systems mentioned in Chapter 2 performs some sort of monitoring of a DHT, they differ in their end-goal of this monitoring as well as how they accomplish it, as does ours from theirs. We do not seek to do an *aggressive* monitoring as with *ClearView*, which only works due to the VUZE-DHT using replication. Instead, we will ensure our node IDs are distributed nicely within the ID-space from the beginning, such that all our nodes do not end up within the same bucket of the other nodes. This is required since we have to rely on passively listening to the traffic, as the Mainline-DHT does not support replication.

Due to the previous discussion, Silverstream more resemble the system *Montra* by Memon et al., however, while the execution might seem similar, the end-goal still differs: As mentioned, they sought to simply monitor the traffic, peers, files, etc., and as such not necessarily keep track of exactly what files are available at any given point in time. This, however, is crucial to us, as our system must facilitate the playback and streaming of said files. Because of these points, we argue that our specific use case has not been investigated much in the academic world and we will therefore need to extensively test the individual parts of Silverstream and its capabilities in regards to looking up files, downloading said files and being able to *flawlessly* play these back.

Most of the existing systems mentioned in 2 depend on peer-to-peer for their streaming. While this is a great use-case and surely what we intend to do as well, it can have issues if too few peers have a given file, causing it to constantly buffer or potentially never being available for playback. For Silverstream to work for end-users, we have to find a way of working around this. A solution could be to play the first song in a playlist while downloading the next. This would solve the issue for all but the first file.

None of the mentioned articles mention downloading or *streaming* from the BitTorrent network which they monitor. Because of this, we have no evidence for any particular method functioning better than others. As such, we have chosen to simply utilise the sequential downloading feature of LibTorrent, which will allow playback to begin before the file has fully finished downloading.

Furthermore, the papers had no mention of creating a database allowing for mapping of actual song names to their corresponding id-hash. We will need to develop this system – ideally also allowing for fuzzy matching or *full text search*. Fuzzy matching is a technique used

when an exact match cannot be found in the database for the text being searched for; this is required because users cannot be trusted with inputting perfectly correct song-names, while at the same time torrents may include misspelled file names. As mentioned, no theoretical proof of any look-up or indexing speeds were found, as such we start from scratch and will have to figure out also what information we need to hold, allowing for quick searches and in the end a nice user experience.

Additionally we also need to be able to scale the system in regards to the number of nodes we insert into the `DHT` in case we need to adjust our search space; this is touched upon by the authors of *Montra*. Last but not least, the `Mainline-DHT` allows nodes to pick their own `node-id`. This means we can further broaden the search space with carefully placed nodes. This is not unlike what *ClearView* does, however we do not intent to jump around the `Mainline-DHT`, but rather place the nodes and simply wait.

To conclude, we establish two main hypotheses:

- Playback should begin within reasonable time
- We should be able to build a database of torrents reasonably quickly

Additionally, we establish some tests we can perform, to make conclusions in regards to the hypotheses.

4 DESIGN AND METHODOLOGY

4.1 DESIGN OF THE SYSTEM AS WELL AS THE EVALUATION

4.1.1 *Evaluations and testing*

We have established a few areas of the system which we wish to evaluate and test. These are listed in the following sections.

4.1.1.1 *Data gathering*

We wish to monitor and evaluate several potential performance bottlenecks with the discovery and collection of data in the `DHT`. It is important that we can gather the required data on the content of the `DHT` in reasonable time, i.e. we cannot afford to wait two weeks for our nodes to get a decent grasp of what is going on in the `Mainline-DHT`. There are generally two different ways we can accomplish this: One way is adding more nodes to the network, which might cause us to take up too much space and run into the issues that Memon et al. ran into while building *Montra*. Because of this we will need to evaluate the number of nodes and optimise the number of nodes. The other way is to set the nodes' IDs to hopefully end up in different buckets of the other nodes in the `Mainline-DHT`, thus causing us to get asked about more files and be exposed to more announcements. Keep in mind, neither of these approaches fixes issues with downloading speeds as they depend on the upload of the other peers seeding the file.

We will also need to test if we can find files and download quickly. To perform most of these evaluations, we will attempt to use the `Mainline-DHT`, since it most accurately resembles a live system, where these aforementioned issues have potential of occurring. However, using `Mainline-DHT` can lead to inconsistencies in our evaluations, as such we would like to additionally run our evaluations with torrents that we host ourselves. This allows us to double-check our results from the `Mainline-DHT`. Whenever *quick enough* or a *desirable* amount of time is mentioned, these refer to the end-user experience and as such will require tests with such people, to determine how long people consider *too long*.

4.1.1.2 *User search network*

Not all of the searching is done in the `Mainline-DHT`. As mentioned, the goal is to have each peer have their own local database mapping

songs to hashes, avoiding a centralised server. An issue, however, with this decentralised setup is that peers need to ask others about a songs, if they do not know about it themselves. We do not consider flooding a solid way of accomplishing this, as it results in a large overhead of messages on top of the already taxing task of participating in the `Mainline-DHT`. Therefore, we would like to establish tests where a user will 1) search for a specific file only known to one user, 2) a file known by many and 3) a file known by none. The reason for each scenario is somewhat self-explanatory, except for the scenario where no one knows the file: This scenario is interesting as we would preferably want to avoid having to question the entire user group, however we need to develop the algorithm allowing for this. Yet again, the time delay we will allow for will be determined through testing with users.

4.1.1.3 *Evaluation framework*

Most of what we seek to evaluate are considered to be related to the `Mainline-DHT`, as we intend on using this as the main way of gathering torrent metadata. However, we also wish to evaluate our searching algorithms in a controlled environment, where we can focus on the amount of time needed for finding a specific song. If we can establish a flooding algorithm for the user search network, we expect the search to function relatively poorly, however, if time allows for us to design an algorithm allowing us to do full text search within our already established network, i.e., without the use of an additional overlay network, it would be interesting to check how much message overhead we avoid as well as how quickly we are capable of locating the correct torrent file. However, as this is pushed to further work, so is the evaluation of said system. Because of this, we will initially evaluate the search and download time of a single user.

4.1.1.4 *Evaluation overview*

To allow for an overview of the evaluations we desire to perform, a table has been made briefly describing in which category we deem the test to be in, which part of the system it evaluates, which requirement there is for the part, which things we can change or vary to change the behaviour of the part as well as whether or not we want to test it on the `Mainline-DHT`. Note, that something is listed as being a mainline test does not mean it will not be evaluated in a controlled environment, it merely means that it will also be tested on `Mainline-DHT`.

Category	Part	Requirement	Method	mainline
Timing related	`BitTorrent` nodes	It shouldn't take too long to crawl `BitTorrent`	vary nodes	✓
	`BitTorrent` nodes	It shouldn't take too long to crawl `BitTorrent`	Vary address space	✓
	Indexer	The indexer should keep up to the infohashes collected	Different algorithms	✗
	User search network	It shouldn't take long for user to find song or conclude nonexistence	Different algorithms	✗
	Whole system	When a user starts searching, download should start relatively soon	Different algorithms	✓
Distance related	User search network	It should not require too many jumps to conclude existence of song	Different algorithms	✗
	`BitTorrent` nodes	It shouldn't take too many jumps to find song from unknown peer	Different algorithms	✗
Data related	`BitTorrent` nodes	There should be a satisfying number of songs in the system	Different algorithms	✓

Figure 1: Overview of desired evaluations

4.2 COMMUNICATING THE DESIGN

4.2.1 *System overview*

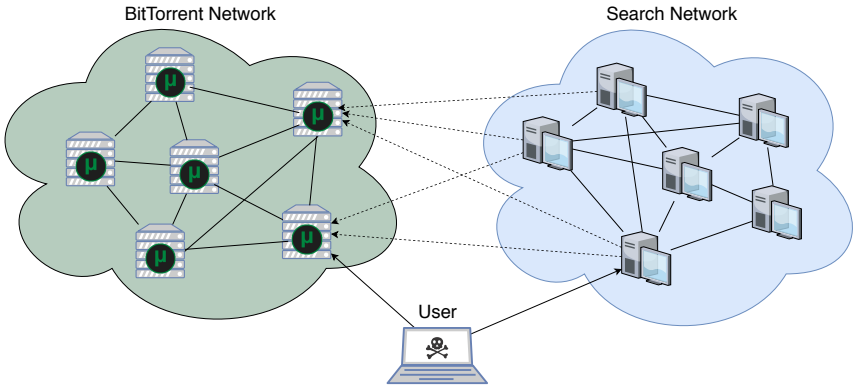

Figure 2: An image describing the system on a very high level

The diagram in figure 2 explains how the user will be simultaneously communicating with the `BitTorrent` `Mainline-DHT` and the `DHT` that we must create between the users of our system to facilitate searching. When a user searches for a song, We have to communicate with the peers of our custom overlay network to find the song if it is not present in the local database, and afterwards with the `BitTorrent` network to actually download the file. The peers of our own search network also have to communicate with the `BitTorrent` network, as they have to crawl and listen to the network as a way of generating their own database. As mentioned, we seek to create a fully decentralised system, as such the peers we insert into the `BitTorrent` network can not all write to a centralised database, as this would create a tracker-like situation. Because of this, we have to let each peer of the search network gain their own understanding of what is in the `BitTorrent` network.

Additionally, to implement the fuzzy matching or full-text search for songs in our search network we expect to implement an overlay network similar to *Cubit*.

4.2.1.1 *Lower level explanation*

We want to implement a system where we insert a certain number of nodes – as determined through testing – into the `Mainline-DHT`. The purpose of these nodes is to passively listen whenever another peer on the `DHT` either announces that they have a specific torrent or calls `GET_PEERS` for a specific infohash; in both cases we will save the specific infohash to a database. An indexer connected to the database will look up the infohashes in the network as they are discovered, gathering metadata about the corresponding torrent such as name, content and known trackers. The indexer will use an extension to the `BitTorrent` protocol known as *BitTorrent Enhancement Proposal 9*, which allows for torrent metadata exchange between peers, allowing us to gather metadata about a torrent using only a few kilobytes of bandwidth. Each user of our system will run multiple indexers concurrently, quickly building an index of available torrents and associated metadata. Allowing the users of the user-search network to search each others' databases will result in a fully decentralised system. To begin with, nodes in the user-Search network will only be able to search their own database, however if time allows for it, we will extend the system to be able to flood the search network in an attempt to find specific files. If we have further time, a full text search will be implemented, inspired by the aforementioned system, *cubit*, but this will be left to further work initially.

4.2.2 *Use cases*

4.2.2.1 *Searching for a song*

The sequence diagram in 3 explains how the system reacts to a user searching for a song. The user initially asks the user-search network – here called `Silverstream-DHT` – if they know any torrent hashes for the specific song. The peer network responds with potential hashes and the user selects the preferred song among the results. To ensure we follow the standards and are able to seed back what we leech, the work of actually downloading the torrent is delegated to the `LibTorrent` framework. `LibTorrent`, however, will only be responsible for the actual `BitTorrent` protocol itself; we will inject known seeders into the framework using our own `Kademlia` implementation. In the end, an `mp3` or some other music file is returned to the user. Additionally, the job of playing the various music formats is delegated to *mpv*, as the details of this process is irrelevant in this course.

4.2.2.2 *Indexing*

The indexer is responsible for populating the database by downloading metadata about the infohashes we have gathered, thereby linking the title of the song, to information about the specific torrent, such

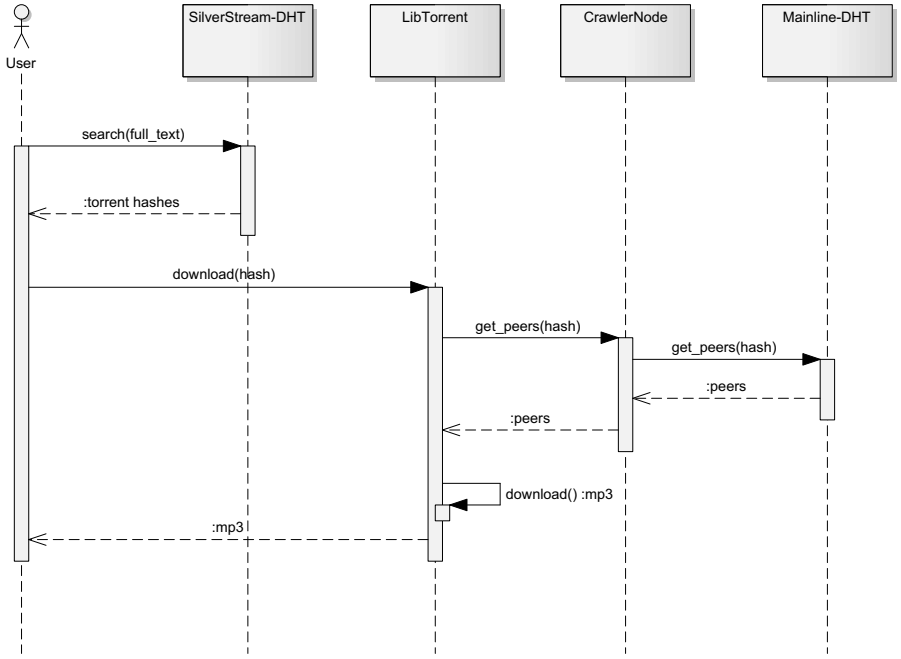

Figure 3: Sequence diagram explaining the process of searching for a song. While the diagram shows an mp3, the indexer supports a wide range of different audio formats.

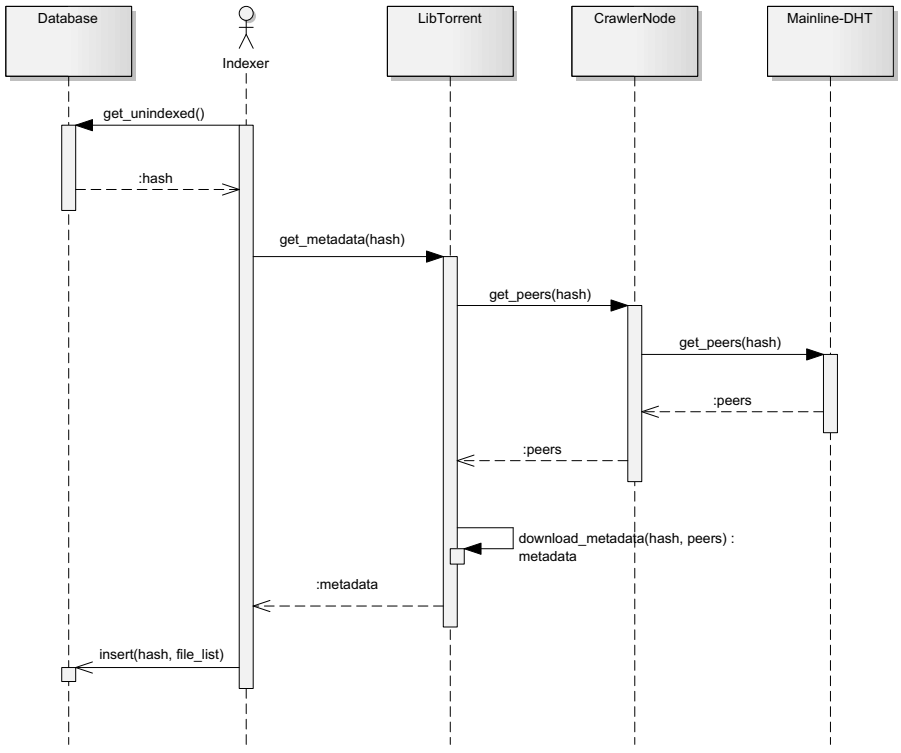

Figure 4: A sequence diagram explaining the process of indexing

as the infohash and the peers who seed it. Through this database, we can allow users in our search network to look up songs based on keywords and find the correct torrent.

5 IMPLEMENTATION

Our system primarily consists of four parts; 1) Crawler nodes, which passively listen in on the Mainline-DHT, 2) an indexer, whose purpose is to download the metadata about the torrents the crawler nodes find and populate the database and 3) the downloading and streaming module.

5.0.0.1 *Kademlia implementation*

Our Kademlia implementation, which drives the crawler nodes, has been designed to allow us to quickly alter the behaviour to accommodate different strategies in regards to the evaluation. As an example, we can specify exactly how many crawler nodes we wish to insert into the Mainline-DHT, how the nodes' IDs are chosen, as well as how infohashes should be saved. The strategy for determining when to save an incoming infohash is a reaction to a BitTorrent Extension Proposal, stating that infohashes should be obfuscated unless the asked node is close enough to the target to realistically hold information about it. This proposal was made to specifically prevent the type of passive listening we are trying to do. Because most BitTorrent software is based on the open source LibTorrent library, we were able to implement an algorithm which attempts to guess when an infohash is obfuscated based on request, destination and our own node-id.

Our strategy for generating the node-id consist of splitting the id-space in a number of sub-intervals equal to the number of nodes which are to be inserted into the Mainline-DHT. We then uniformly select each node-id from one of these sub-intervals. We theorise this should ensure an even distribution of node-ids and thus allow the nodes to be inserted into as many different buckets as possible.

Our implementation fully conforms to the specification as laid out by BitTorrent Extension Proposal 5, including the ability to announce our own IP and port to an infohash. While this is not particularly useful for crawling the DHT, we believe it could become important for future work, as it could serve as a way to bootstrap our own overlay network without relying more centralised seed nodes. Specifically, we could chose a fixed 160-bit string all Silverstream clients would announce their address to, allowing discovery of other nodes in a fully decentralised manner.

5.0.0.2 *Indexer implementation*

The indexer is implemented as a `sqlite` database. We chose this database engine because it is lightweight, does not require additional software, and supports full text searching by default. The indexer runs a number of worker processes in an event loop, continually pulling out an unindexed infohash and associated peer-list from the database. For each infohash, it requests the metadata of the torrent file from the peers in the peer-list, as described in the sequence diagram in figure 4. Once the metadata has been retrieved, the indexer will clean the file names and insert it into the database, creating a searchable database, mapping keywords to infohashes. If the torrent contains an album or file-tree of multiple files, each file path is cleaned and combined to allow the user to search for any song in the album, as well as albums themselves. Additionally, the indexer will filter any files that are not audio files, such as `.mp4` or `.mov`.

5.0.0.3 *LibTorrent*

`LibTorrent` is a popular torrent framework which forms the basis of most torrent programs such as *Deluge* and *qBittorrent*. Using a pre-existing library ensures that we will be participating properly in the swarm, seeding back what we download. `LibTorrent` allows us to implement partial downloading, such that we can download a single file or track from an entire album. Furthermore, by utilising the sequential downloading feature of `LibTorrent`, which ensures the content of the file is downloading in correct order, we can begin audio playback before the file has fully finished downloading – essentially enabling music streaming with a permanent buffer. Note, however, that some music formats does not support playback of a partially downloaded file. For this reason, we have developed a heuristic to determine when a song ready for playback that takes into account the limitations of the audio format. The calculations also include some estimates on the bit-rate of different audio formats, allowing us to begin playback of a compressed `mp3` sooner than a lossless `flac`, for example. These calculations are based the current download rate, the remaining download time, and estimations of bit-rate and average song duration. These features help minimise the playback-delay experienced by the end-user, allowing for a smoother experience. To further decrease the delay, we also immediately connect to peers we know about from the crawling of the `DHT`; if these peers do not provide a satisfactory download speed, we will request more peers from one of our `DHT` crawler nodes. To increase lookup performance, this node is selected as the closest to the torrent's infohash, since nodes in `Kademlia` know more about their immediate neighbourhood.

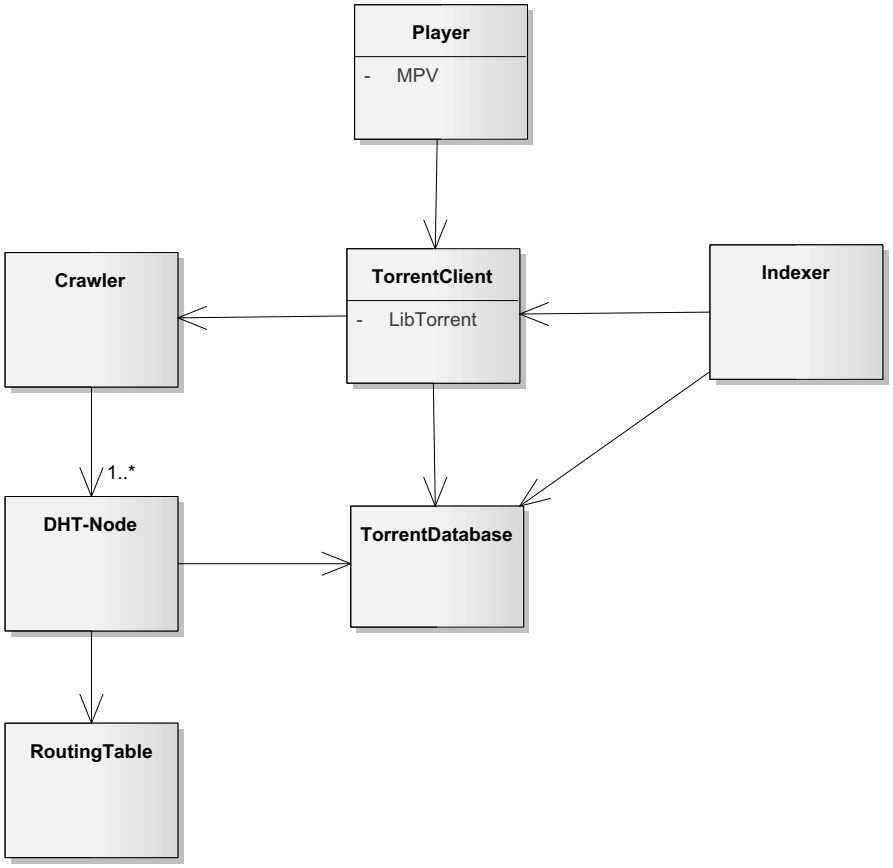

Figure 5: Overview of the implementation. The `Player` and `TorrentClient` objects use a limited set of functions from the `MPV` and `LibTorrent` libraries, respectively.

5.0.0.4 *Final system overview*

The diagram in figure 5 detail how the main components of `Silverstream` are connected. The command-line interface has been intentionally left out, since including it would be messy as it talks to most of the components, allowing the user to control them. Note that the crawler contains many `DHT Kademlia` nodes; the diagram depicts a single one with its routing table and knowledge of the single torrent database, which is shared among all components.

When the user requests playback of a specific song from an infohash the CLI queries the `Player` component, which in turn starts a download in the `TorrentClient`. The player will continuously query the torrent client for its download status until it meets the requirements for playback as per our heuristic, as previously detailed.

While `BitTorrent` Extension Proposal 5 goes into great detail how `DHT` nodes should communicate, it is vague on how node should behave locally. For this reason, we have chosen to follow the `Kademlia` paper closely, taking special care to follow the advice given in the

implementation-section, which, among other things, describe how to increase the performance by utilising a replacement cache.

5.0.0.5 *Conclusion on implementation*

Although we did not fully realise the desired design, we have built a system on which we are capable of evaluating the majority of our tests. We successfully implemented a custom `Kademlia` node that can communicate with the mainline `DHT`, we built an indexer that asynchronously and efficiently fetches metadata, allowing users to perform full text searches, and we have a custom torrent client tailored for music streaming based on the `LibTorrent` framework, which allows sequential as well as partial downloads.

5.0.1 *Further work*

Future work could consist of implementing a full-text search capable user search network based on Cubit, as explained in chapter 2. The addition this network would make it possible for us to evaluate algorithms for calculating distances from keywords but also key phrases. The latter is more difficult, since the obvious choice would be the *Levenshtein distance* for keywords, however this isn't defined for key phrases.

6 EVALUATION

We here list our two main hypothesis', which we wish to evaluate, as well as the results of the tests we performed to do this evaluation.

6.1 PLAYBACK SHOULD BEGIN WITHIN REASONABLE TIME

In an effort to precisely define "reasonable time", we asked users to compare our system to other popular streaming platforms, specifically *Spotify* and *Apple Music*. We found that the centralised solutions would begin playback almost immediately, while our system would generally take between 5 and 10 seconds to start playback, depending on the file type and popularity of the torrent. In general, users did prefer the faster playback for the first song, however, when enqueueing the next song in most cases users did not report any annoyances with our system. In some cases, however, we did find that the system – due to its reliance on the `BitTorrent` network and its seeders – would stall either from the very beginning or halfway through downloading a song.

By starting the playback before the files had fully downloaded using our heuristics, we managed to significantly cut down waiting time in most cases. Unfortunately, all downloads are still reliant on the availability of seeders, and as such we found that the time of day and nationality of torrent would sometimes play a crucial role. As an example, we found that Russian Top-lists were very well-seeded during the day, while downloading late in the evening would sometimes fail. Therefore, to accomplish an evaluation of the general playback delay, we chose torrents that were both international and somewhat popular to make it more likely that it would be well-seeded.

As our database only contains a mapping from songs to infohashes, when a user selects a song for playback, the system must re-download the relevant metadata even though it has already fetched it as part of the indexing process. Instead, we test if it provides any significant speedup in the playback delay to cache the `.torrent` files themselves as well. Our hypothesis is that it should result in a speedup equal to the time it takes to perform a successful metadata exchange.

6.1.1 *Evaluation and results*

Our hypothesis was confirmed, as we, on average, were able to start playback one second earlier with the `.torrent` file cached. However,

	With `.torrent` file	Without `.torrent` file
With streaming	08.05 seconds	09.75 seconds
Without streaming	17.1 seconds	18.05 seconds

Table 1: Table describing the average playback delay of randomly selected songs from international torrents.

while the process of saving the metadata to a `.torrent` file only took 1.4ms on average for the indexer, the caching of a large number of `.torrent` files do take up quite a bit of space. As of performing this evaluation, we have 11.083 `.torrent` files in our index, taking up a total of `420MiB` distributed over 126.257 songs, leading to an average song taking up `3.5KiB` of space in `.torrent` files. However, as shown in table 1, the playback delay can be decreased by roughly 25% by not having to fetch the metadata for every playback. We conclude that the metadata exchange protocol takes on average one second to complete; a second that constitutes a major portion of the total delay when streaming is enabled, and thus caching the `.torrent` file may be a worthwhile investment for many users. As this is a time versus space trade-off, we implemented a switch that allows the user to choose on system start up.

Additionally, we observed that `.flac` files take orders of magnitude longer to download than ordinary `.mp3` files. This issue became obvious when we performed the tests without the streaming-feature enabled. In general, `.flac` files took on average *25 seconds* to fully complete. As all experiments throughout our evaluation were performed on both `.mp3` and `.flac` files, this is what caused the average playback delay to increase as much as it did. On the other hand, the difference between streaming and non-streaming `.mp3` files were much less, as these files were generally not more than a few megabytes in size. Furthermore, the difference between `.torrent` and non-`.torrent` in the case without streaming was negligible, as we yet again had to wait for `.flac` files to fully download, this completely overshadowed the time it took to download the metadata.

6.2 WE SHOULD BE ABLE TO BUILD A DATABASE OF TORRENTS REASONABLY QUICKLY

We wish to gather a database of all popular songs for the user to search through. To accomplish this, we have a few different variables we can change, including the number of crawler nodes and the `node-id` generation algorithm. We expect the number of torrents found to scale linearly with the number of peers we insert into the `Mainline-DHT`, and we expect an algorithm that chooses `node-IDs` such that they are spread throughout the ID space to perform better than a uniformly random one.

6.2.1 *Evaluation and results*

To find the best combination of the variables, we established seven different servers all running with a different number of `crawler-nodes`, in the same geographical region at the same time, due to the time-sensitive nature of the `Mainline-DHT` network.

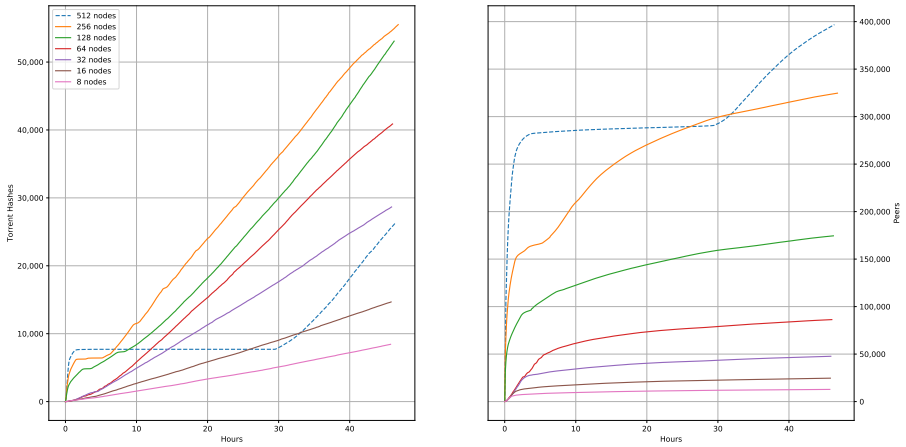

Figure 6: The number of found torrent hashes and peers in routing table as a function of the number of crawler nodes.

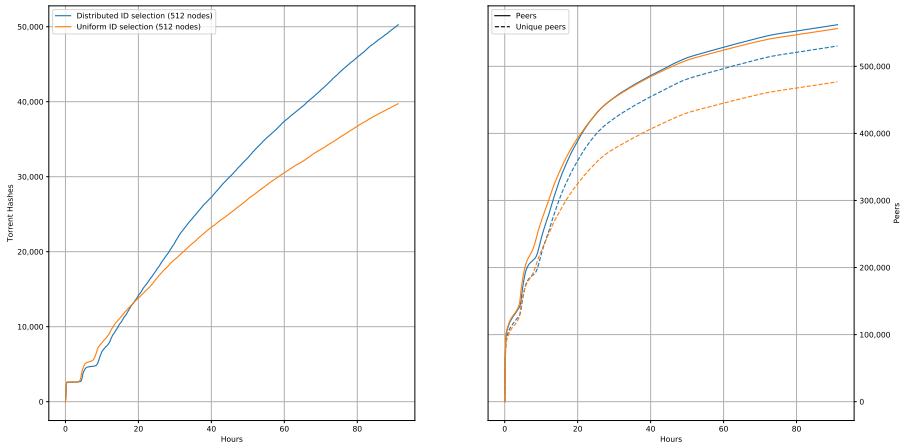

Figure 7: The number of found torrent hashes and (unique) peers as a function of the number of crawler nodes, using two different strategies for generating node IDs.

As mentioned in related work, some effort has been put in to determining the appropriate number of nodes to insert to optimise the number of discovered torrents. However. as we did not have the compute power to run 32.000 nodes, we had to stop at 512, which is already cutting it close to malfunctioning, as is seen in figure 6. Because of this, our initial hypothesis about hashes scaling with the amount of nodes, ends up being incorrect. However, we estimate that the server running 512 nodes would catch up eventually, as it quickly overtakes the server running 256 nodes in the number of known peers. The

malfunctioning was largely due to a bottleneck on the CPU of the server, and we estimate that running the experiments on more powerful servers would result in more consistent results.

Apart from these issues, our hypotheses is correct, as the number of torrent hashes found clearly scales well with the number of nodes initiated. Additionally, we find that the number of peers in the routing table turns out to be logarithmic. This means we could essentially have saved CPU cycles by not adding any more peers after a certain timestamp. By looking at the right graph of figure 6, we can see that the server running with 128 nodes almost performs as well as the server running with 256 nodes, while the server with 128 nodes knows half as many peers. This leads us to conclude that we could have stopped adding peers to the routing table on the server with 256 nodes, almost instantly, which would have freed up CPU cycles for this server. We can not exclude that the performance for the server with 256 nodes, is not due to hardware limitations as well, however we hypothesise that if this is the issue, freeing up CPU cycles by adding peers to the routing table less aggressively could have greatly benefited this server.

Figure 7 confirms our hypothesis that an algorithm that chooses `node-ID`s such that they are spread throughout the ID space performs better than a uniformly random one; in particular, we notice that the number of nodes present in both routing tables after 90 hours is almost exactly the same, while, more importantly, the number of unique peers is significantly higher in the system running the distributed ID-selection algorithm. This is expressed in the left-hand graph, which shows that the number of found torrents is significantly higher in the system running this algorithm.

7 CONCLUSION

We have created a fully decentralised music streaming platform, `Silverstream`, which is capable of streaming directly from the `Mainline-DHT`. To accomplish this, we have implemented a `Mainline-DHT` crawler with a custom `Kademlia` protocol, a framework built on top of `LibTorrent` and an indexer, using the aforementioned `LibTorrent` framework, to built a full text search-able database over all the songs we know of. We have provided a concise description of which components communicate together and how these individual components have been built. We have avoided going into too much technical detail, as we deemed this unnecessary for the understanding of the system as a whole.

We have experimented with different ways of implementing the `Mainline-DHT` crawler, by varying how its nodes have their `node-ids` determined and how many nodes it should have. We found that the way of generating `node-ids` in general performed better, when compared to randomly selecting the `node-ids`. Additionally, we found that there is a linear correlation between the number of nodes the crawler has and how many `.torrent` files it finds.

We also experimented with the implementation of the `LibTorrent` framework and found that while being able to stream music, contrary to having to wait for the download to finish, yields a significantly faster playback in general, the gain from it is much higher when the music quality is higher, e.g. for `.flac` files over `.mp3` files.

Furthermore, we tested how the `indexer` should function and made an argument for why someone might apply the switch to save all the `.torrent` files, as the indexer has to fetch the metadata for each infohash regardless, whenever it wishes to *index* a file. The main point is, that we see it saves approximately 25% time in regards to playback, when we tested it on a bunch of random files existing in the `Mainline-DHT`.

BIBLIOGRAPHY

[1] Scott A. Crosby and Dan S. Wallach. *An Analysis of BitTorrent's Two Kademlia-Based DHTs*. URL: https://scholarship.rice.edu/bitstream/handle/1911/96357/TR07-04.pdf?sequence=1&isAllowed=y.

[2] Guy Douglas. *Copyright and Peer-To-Peer Music File Sharing: The Napster Case and the Argument Against Legislative Reform*. 2004. URL: http://www.murdoch.edu.au/elaw/issues/v11n1/douglas111.html.

[3] Stephanie Mlot. *'Popcorn Time' Is Like Netflix for Pirated Movies*. Tech. rep. PCMAG, 2014. URL: https://uk.pcmag.com/internet-3/10462/popcorn-time-is-like-netflix-for-pirated-movies.

[4] Andrew Loewenstern & Arvid Norberg. *DHT Protocol*. Bittorrent.org, 2008. URL: http://bittorrent.org/beps/bep_0005.html.

[5] Ghulam Memon & Reza Rejaie. *Large-Scale Monitoring of DHT Traffic*. URL: https://www.usenix.org/legacy/events/iptps09/tech/full_papers/memon/memon_html/.

[6] Wikipedia contributors. *Countries blocking access to The Pirate Bay — Wikipedia, The Free Encyclopedia*. [Online; accessed 4-November-2018]. 2018. URL: https://en.wikipedia.org/w/index.php?title=Countries_blocking_access_to_The_Pirate_Bay&oldid=864920733.

[7] Scott Wolchok and J. Alex Halderman. *Crawling BitTorrent DHTs for Fun and Profit*. URL: https://www.usenix.org/legacy/event/woot10/tech/full_papers/Wolchok.pdf.