

# Advanced distributed systems

## 5DV205

### Fall, 2024

Project report - SPADGIFY  
Version 1.0

Name:	Anton Dacklin Gaied	Sinthujan Ponnampalam
CS-id:	ens19ann	c20spm
UMU-id	anda0220	sipo0008

Teacher: Per-Olov Östberg

Department of Computer Science  
Umeå University  
November 3, 2024

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Usage</b>	<b>1</b>
2.1	The node . . . . .	1
2.2	The test client . . . . .	2
2.3	The React client . . . . .	3
<b>3</b>	<b>Chord</b>	<b>5</b>
<b>4</b>	<b>gRPC</b>	<b>5</b>
<b>5</b>	<b>Caching</b>	<b>6</b>
<b>6</b>	<b>Web socket</b>	<b>6</b>
<b>7</b>	<b>REST</b>	<b>6</b>
<b>8</b>	<b>Node</b>	<b>7</b>
<b>9</b>	<b>Testing</b>	<b>7</b>
9.1	gRPC Tests . . . . .	7
9.1.1	Logic Tests . . . . .	7
9.1.2	Performance tests . . . . .	8
9.2	Streaming Tests . . . . .	8
9.2.1	Logic Tests . . . . .	9
9.2.2	Performance Tests . . . . .	9
<b>10</b>	<b>Result</b>	<b>10</b>
<b>11</b>	<b>Conclusion</b>	<b>18</b>

# 1 Introduction

The goal of this project was to build a distributed system that mimics the functionality of *Spotify* [2]. As a tribute to the original creators of Spotify the name of this project is *Spadgify* which is a concatenation of the members' initials (Sinthujan Ponnampalam = 'Sp' and Anton Dacklin Gaied = 'adg') with the addition of 'ify'.

For this project, several different technologies were used to achieve this functionality: *Chord*, *gRPC*, *REST* and *Web sockets*. The implementation consists of several nodes that use the Chord protocol to distribute data in the cluster. In addition, a client written in *React* is also provided such that a user can connect to an arbitrary node and listen to music available somewhere inside the cluster.

All tests in this report were performed with two nodes running on a laptop and one node running in a VM on a Proxmox server running on the local network. A test client that is also running on the laptop is used to measure the performance of the cluster by connecting to the node running on the server and performing a series of tests. The computer used is a laptop with 8 cores without Ethernet port, hence WIFI was used to connect to the local network. The server is connected to the router with an Ethernet cable.

The tests aimed to measure the retrieval time of music files between nodes as well as measuring the streaming capabilities of the cluster. The goals set for this project were to be able to run at least three nodes in the cluster and be able to support 100 clients at the same time.

The project was written using Java 18.0.1.1, Maven 3.9.9 and libprotoc 28.0.

## 2 Usage

This project has three parts: the node implementation, the React client and a test client.

### 2.1 The node

The node uses the following synopsis:

```
Usage: chord [-h] [-c=<cache>] [-d=<delay>] [-i=<remoteIp>]  
           [-p=<remotePort>] mode port m
```

where:

- *-h* is help.
- *-c* is the cache size of a node.
- *-d* is the delay at which the node runs its stabilize algorithm.
- *-i* the IP of a node in a cluster that this node wants to join.
- *-p* the port of a node in a cluster that this node wants to join.
- *mode* 1 for creating a new cluster or 0 for joining a cluster.
- *port* the port that the node should use for communicating with other nodes.
- *m* the m-bit identifier used by the Chord protocol to hash.

To build the node part do the following (**Using IntelliJ IDEA.**):

1. Unzip zip file.
2. Open the project inside the "SPADGIFY" folder as a Maven project.
3. Perform "Maven install"
4. In the folder "Target" there will be a *Jar* file named "spadgify-1.0-SNAPSHOT.jar" which is the node part.
5. To create a new cluster:
  - (a) Run jar through a terminal with: `java -jar spadgify-1.0-SNAPSHOT.jar -c 10 -d 1000 1 8185 3`
6. To join a cluster:
  - (a) Run jar through a terminal with: `java -jar spadgify-1.0-SNAPSHOT.jar -c 10 -d 1000 -p 8185 -i 192.168.38.126 0 8187 3`
7. **The nodes will show their finger tables and what ports are used for the web socket (default 8080) and REST server (default 8000) in the terminal as well as their successors and predecessors.**
8. To perform a graceful exit of a node, use the `ctrl + c` command in the terminal.

## 2.2 The test client

Inside the Java project there is also a test client for testing the cluster. It uses the following synopsis:

Usage: client [-hlpV] [-u=<pathToMusic>] nodeIp nodePort m socketIp  
socketPort restPort

where:

- *-h* is help.
- *-l* tells the client to do the logic tests.
- *-p* tells the client to do performance tests.
- *nodeIp* is the IP of the node that this client want to connect to.
- *nodePort* is the port of the node that this client want to connect to.
- *m* the m-bit identifier used by the Chord protocol to hash.
- *socketIp* is the IP of the socket that this client want to stream music from.
- *socketPort* is the port of the socket that this client want to stream music from.
- *restPort* is the port of the REST server running parallel to the socket. This is used to fetch what content that is available in the cluster.

Open the file *TestClient.java* in IntelliJ and right click the green run button, select "modify run configurations". Upload music to the cluster by entering:

```
-u ../../../../testMedia/input-music 192.168.38.126 8185 3 192.168.38.126 8080 8000
```

or run logic and performance test by entering:

```
-l -p 192.168.38.126 8185 3 192.168.38.126 8080 8000
```


in the program arguments section. Worth noting is that the performance tests uses dummy data that can not be streamed by the React client.

## 2.3 The React client

To use the React client:

1. Navigate to the folder *react/my-spadgify-app*
2. Perform *npm install*
3. Perform *npm run dev*
4. This will start the client at a URL provided in the terminal.
5. In the client, enter the IP:Port combination of the socket and rest server to connect to. Figures 1 and 2 show how this is done.

6. Click on a song in the table and then click the play button to start listening and click the stop button to stop listening. A user also has the ability to search in the table using the search bar.

 Spadgify

192.168.38.126:8080

Socket server: 192.168.38.126:8080

192.168.38.126:8000

Rest server: 192.168.38.126:8000


Init player

Search...

Song	Artist	Album
------	--------	-------

© 2024 Spadgify - music for some

Figure 1: Connect to web socket and rest server.

 Spadgify

Requested song:

Search...

Song	Artist	Album
tdvrsygdolafwua	ajrvmsivefhyhc	phirtzyuabzbbxd
lganwtumkcynt	qpvikzslrgznqp	xeudnzsqhipifga
vawzwzlfqjkiur	mhhapntefbjhpf	mgoqttbzbgozeyc
trhtuikihvvywf	frmkfgesyorhjr	pwtenlwvpczowuh
arpnekdzfjxvrrl	owqjrethnnuqayu	ccasnmvapczrjcc
ltlakanjqervnlf	xlptbkwwxdgfmnm	tatthbsmnbeonvl

▶

■

© 2024 Spadgify - music for some

Figure 2: Connected and the content of the cluster is shown in the table.

### 3 Chord

To distribute the data on our nodes we have used the Chord protocol. Chord is a key-based peer-to-peer protocol used in distributed systems to efficiently locate nodes that store specific pieces of data, in our case .mp3 files. The goal of the Chord system is to enable scalable and decentralized data lookup by distributing and balancing data across multiple nodes. Chord is built on a distributed hash table (DHT), where nodes in a network collectively manage a set of key-value pairs. In Chord:

- Each piece of data (or resource) is associated with a key.
- Each node in the network is assigned a unique ID. We used the IP and port to calculate the ID for a node.
- Chord uses consistent hashing to map both keys and nodes into the same circular identifier space.

Chord is designed to find which node stores a particular key. The key is typically a hash value of the data, but in our implementation, the key is a hash value of "song name - artist name - album name" and the value is the .mp3 stored as a byte array. The lookup algorithm in Chord allows any node in the system to efficiently find the node responsible for a given key.

Our Chord implementation also handles dynamic changes in the network, such as nodes joining or leaving. When a new node joins, it receives responsibility for a portion of the key space from an existing node. When a node leaves its keys are distributed to its successor.

The main reason why we went with this approach was that we found the second assignment interesting but we didn't have time to solve it, therefore we integrated it into our project. For a more detailed explanation of the Chord protocol, we refer to the Wikipedia page [3] or the original paper [1], both were used to implement the Chord protocol in our project.

### 4 gRPC

Communication between the nodes is done using gRPC [4], which is a high-performance, open-source **remote procedure call (RPC)** framework developed by Google. It enables different services or applications to communicate with each other over a network, often between servers or between clients and servers, in a fast and efficient way. gRPC is based on the RPC model, where a client can call methods or functions on a remote server as if they were local:

- The client sends a request to the server to invoke a function.
- The server processes the request and sends back the result.

gRPC uses **Protocol Buffers (Protobuf)**, a binary serialization format, for defining the data structure and serializing messages. In gRPC, services and their methods are defined in the `.proto` file, an important service method of gRPC is streaming, which is what we used to send files between nodes. Moreover, we use two types of services:

- One node service that handled the Chord protocol, i.e. by making sure nodes could join, leave, update finger tables etc.
- One file service that handled the transmission of keys and data between nodes.

## 5 Caching

In our system, we have implemented a read-through cache. In a read-through cache, the emphasis is on efficiently loading and serving read-heavy data after the initial request. This was particularly useful in our application since in a real-world scenario only a fraction of the total content is in high demand and by using a read-through cache we could eliminate the time spent on either reading the highly requested content from disc to memory or requesting the file from another node over the network.

## 6 Web socket

In order to provide the ability for clients to stream music from the cluster the nodes implement a web socket server. This web socket server listens to incoming connections from clients. If a client connects, the server assigns the connection to a worker thread that will return the data sought after by the client.

## 7 REST

The system contains a very simple REST API [5] with a single endpoint which the React client uses to retrieve all songs that exist in the network. The songs are displayed in the user interface.



## 8 Node

Using the technologies described in Sections 3, 4, 6 and 7 the creation of the *node* was possible.

The node is the core of this project. The node connects to other nodes as described in Section 3 and can retrieve data from other nodes when joining a cluster and push data onto its successor when leaving a cluster.

On startup, the node creates two folders: *logs-spadgify* and *media-spadgify* respectively. The node logs events into a log file stored within the first folder and stores files that the node is responsible for in the second folder.

When leaving a cluster the node passes on the responsibility of its key space to its successor and transfers the files, removes its own files from disc, logs the last events and shuts down.

## 9 Testing

Testing of the system has been done, though limited by environment and hardware. A combination of logic and performance tests has been run to test the functionality of the system to evaluate its performance and to find potential bottlenecks.

### 9.1 gRPC Tests

The gRPC tests were divided into logic and performance tests.

#### 9.1.1 Logic Tests

The first set of tests test the logic of our Chord implementation to make sure data can be stored and retrieved properly. The three logic tests were:

- **List all songs:**
  1. List all songs in the network before storing, expect to list 0 songs
  2. Store/upload a song to a node
  3. List all songs in the network, expect to list 1 song
  4. Delete the song
  5. List all songs in the network, expect to list 0 songs
- **Store and delete:**

1. Send a request to retrieve a specified song from a node, expect to retrieve none
  2. Store a song in the node
  3. Send a request to retrieve the stored song, expect to retrieve the song
  4. Delete the song from the node
  5. Send a request to retrieve the deleted song, expect to retrieve none
- **Store duplicate:**
    1. List all songs in the network, expect to list 0 songs
    2. Store the same song twice
    3. List all songs in the network, expect to list 1 song
    4. Store another song
    5. List all songs in the network, expect to list 2 songs
    6. Delete both songs
    7. List all songs in the network, expect to list 0 songs

### 9.1.2 Performance tests

The performance tests of our Chord implementation measured the performance of the communication between the nodes by making sure the requests were directed to a node not running on the same machine as the node making the requests. The test was run 2 times, first without any caching of songs, and the second time with caching. The songs were first created and uploaded to the cluster, and then we measured the time it took to retrieve the songs, this was then repeated 10 times and the results were presented as box plots. Running without cache in contrast to running with cache differed in the way the songs were created. When we tested without the cache, each song was unique and only requested once, therefore the cache was never utilized since each song was only requested once. However, when tested with the cache we requested a number of songs (half of the cache size) multiple times, therefore the cache was utilized.

## 9.2 Streaming Tests

To test the system from a client's perspective we used a web socket server.

### 9.2.1 Logic Tests

Before the logic tests were run, songs were uploaded to the cluster and after the tests they were deleted. We had two logic tests, one to test the REST API endpoint and one to make sure it was possible to stream a song. The first test simply sent a GET Request and checked that the response contained all the correct keys such as song, duration of the song, artist, etc. The second test opened a connection and made a request to stream a specified song. If the response was `null`, the test had failed.

### 9.2.2 Performance Tests

The streaming performance was tested by making a growing amount of clients (1,2,4,8) simultaneously make retrieve requests to the Chord network. This was done to simulate a real-world situation where multiple end-users use the system at the same time, unfortunately, we were limited by our hardware which was why we only could test 8 clients simultaneously. We ran two tests, one that measured the total time it took to retrieve all the songs and a second test that measured how long each request took for each client. The size of the songs was 5MB, and we also measured the time each request took when requesting only 5% of a song since that is usually how much is needed before you can start playing the song from a client's perspective, and as you stream the song the rest of the song is retrieved.

The purpose of the tests was to see how many clients can simultaneously stream music, this way we would see at what point the system breaks and where the hardware becomes a limit. If a song is 180 seconds long, and a client continuously retrieves 5% of the song then that retrieval time needs to be less than 5% of 180 seconds which is 9 seconds. If at some point the retrieval time is longer, due to many clients streaming at the same time, our system has reached a breaking point.

The two tests explained in more detail:

- **Total retrieval time:**
  1. Store all songs
  2. Measure the total time it took to retrieve all songs for 1,2,4,8 clients respectively
  3. Remove the songs
- **Per client request time:**
  1. Store all songs

2. Measure the time it took for each individual request for 1,2,4,8 clients respectively
3. Remove all songs

## 10 Result

The results from the tests can be seen in the figures below. Note that a *task* is a retrieve of a .mp3 file, either the whole file or 5 % of the file and that all clients are making their requests to a node in parallel. All clients are connecting to a single node running on another machine than the clients. The cluster consists of three nodes running on different machines. When the caches are turned on the size of the cache in each node is set to 10. Lastly, the files inside the cluster were randomized for each iteration of a test, hence the distribution of the files was different for each test. In one test node A might have 10 out of 16 songs and nodes B and C are storing the remaining 6. In the next test node A might have 3 songs instead. The clients' shared bandwidth is  $\approx 70$  Mbit/s and the tests performed in Figure 3 and 4 used one thread to simulate the behaviour of a node requesting data from other nodes in a cluster.

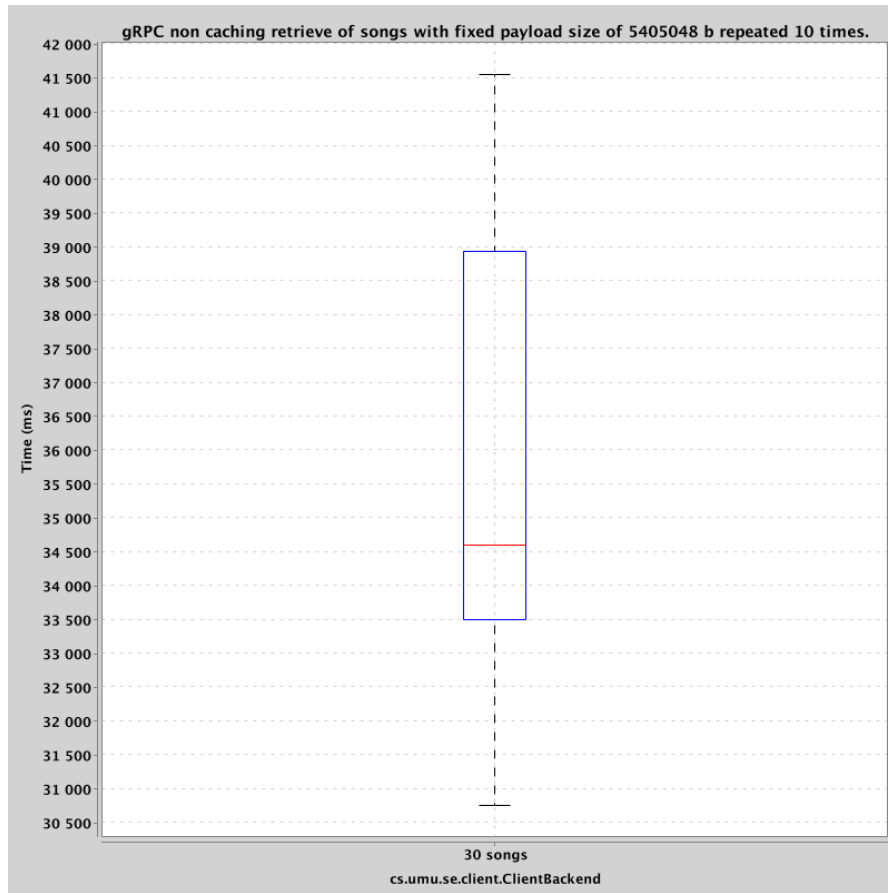


Figure 3: The result when doing retrieves of 5 Mb files from the cluster using the nodes' own gRPC communication protocol with caches turned off.

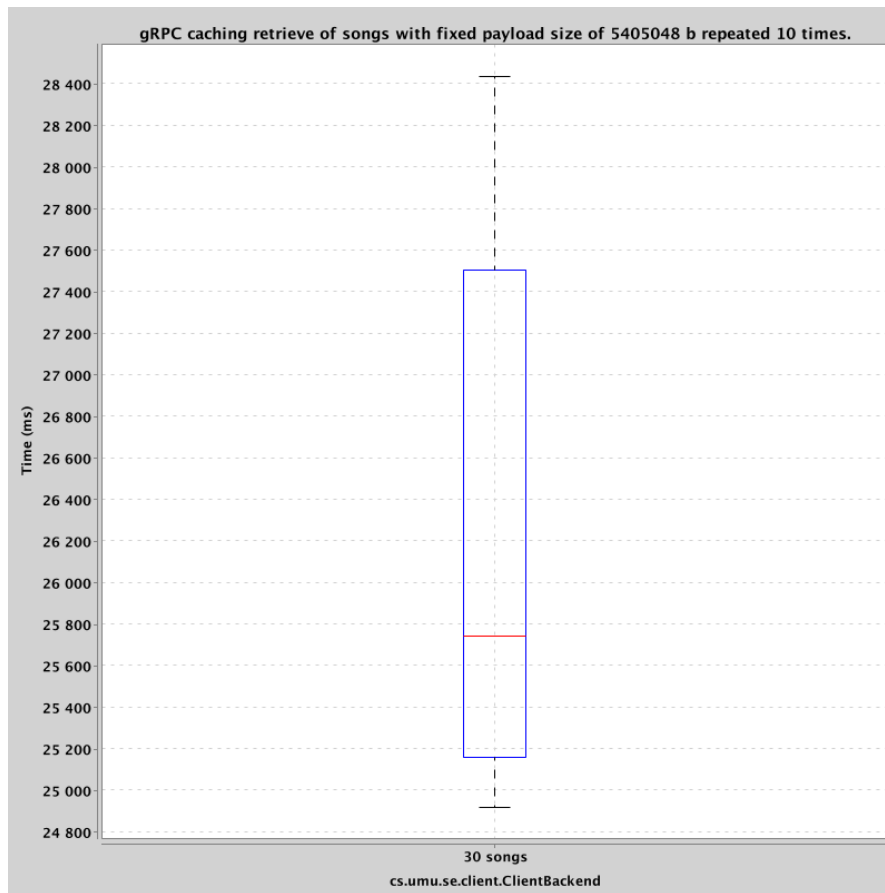


Figure 4: The result when doing retrieves of 5 Mb files from the cluster using the nodes' own gRPC communication protocol with caches turned on.

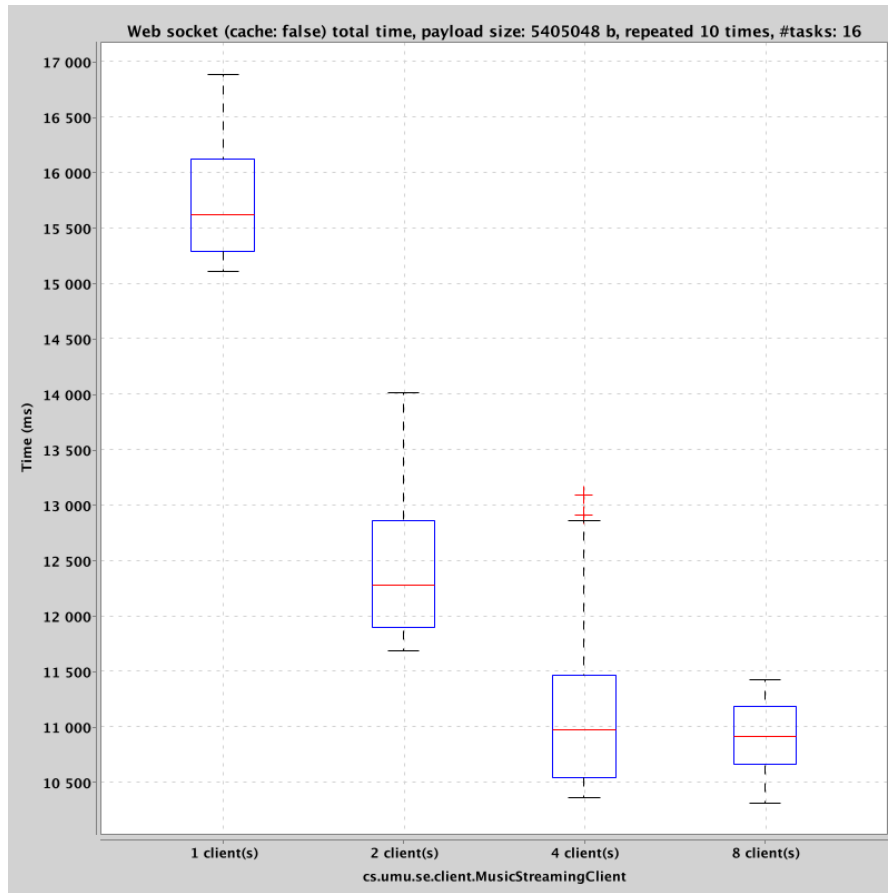


Figure 5: The result when measuring the total time it takes  $C$  clients to retrieve 16 .mp3 files with the caches turned off.

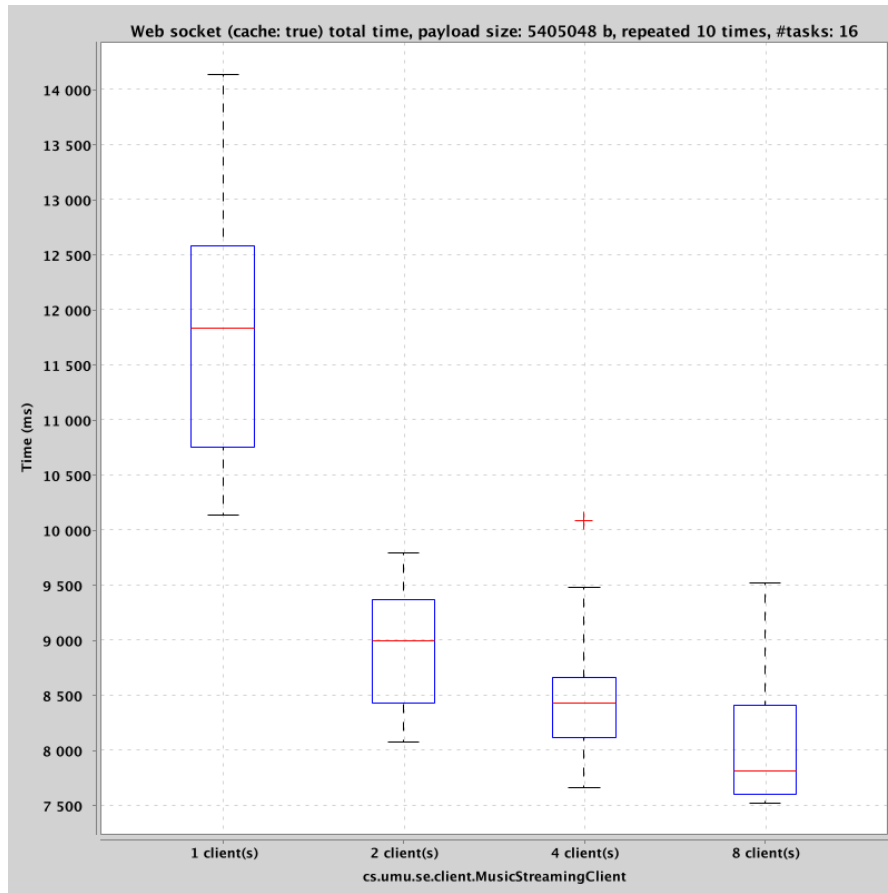


Figure 6: The result when measuring the total time it takes  $C$  clients to retrieve 16 .mp3 files with the caches turned on.



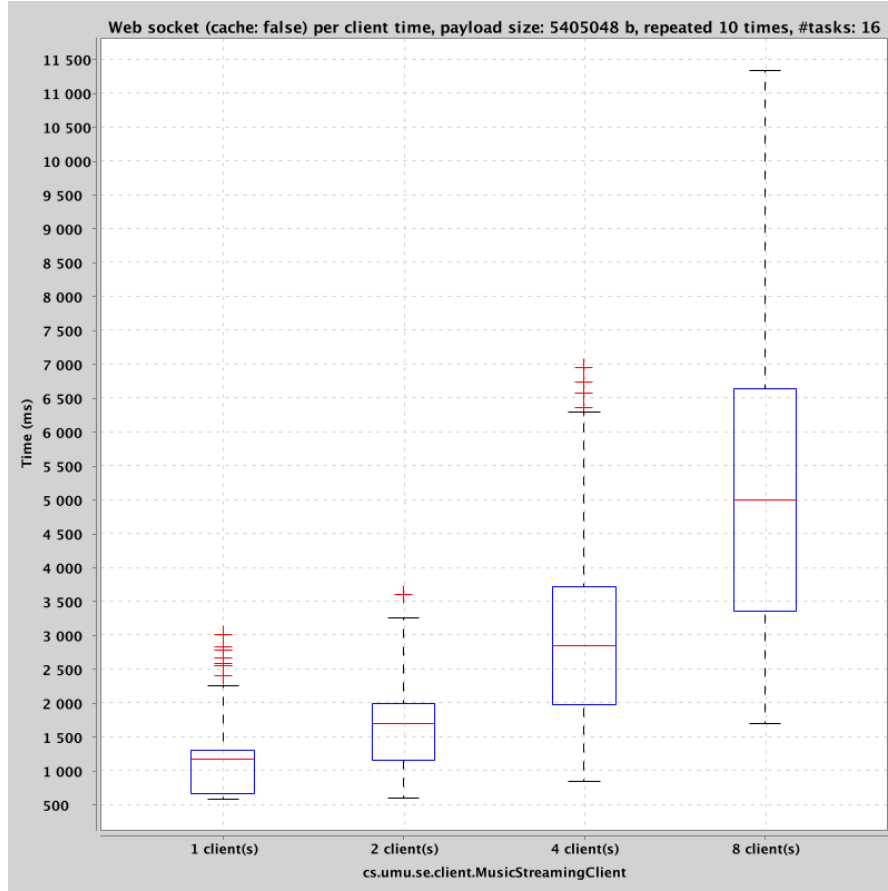


Figure 7: The result when measuring the per retrieve time when  $C$  clients are retrieving a whole .mp3 file through a node's web socket with the caches in nodes turned off.

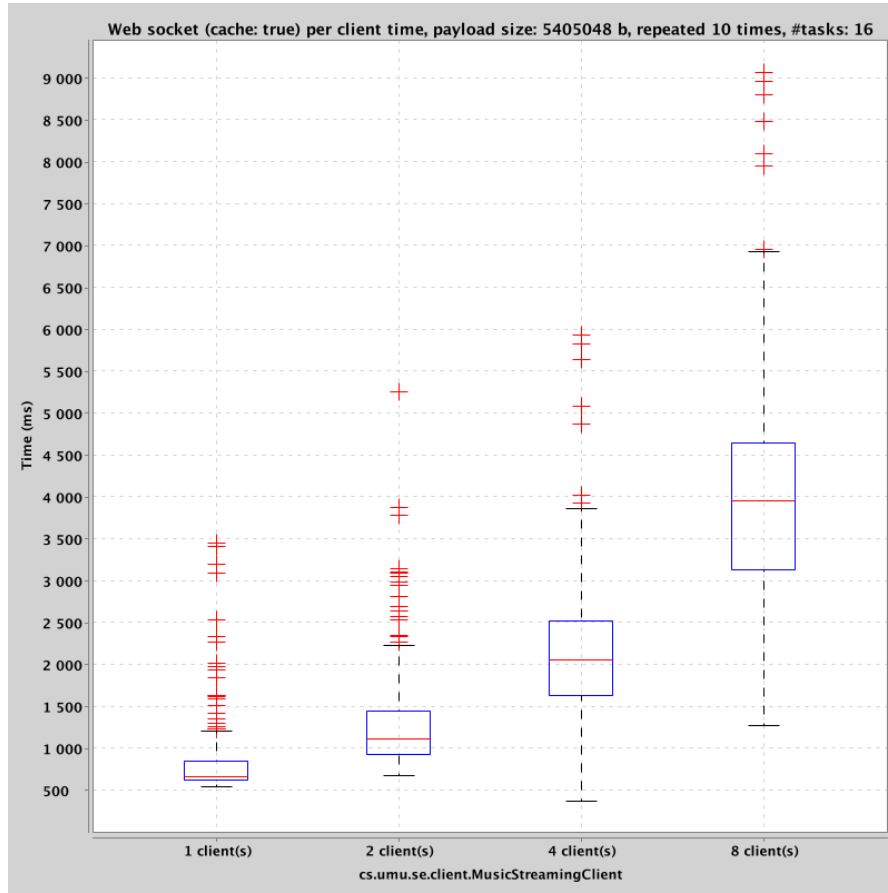


Figure 8: The result when measuring the per retrieve time when  $C$  clients are retrieving a whole .mp3 file through a node's web socket with the caches in nodes turned on.

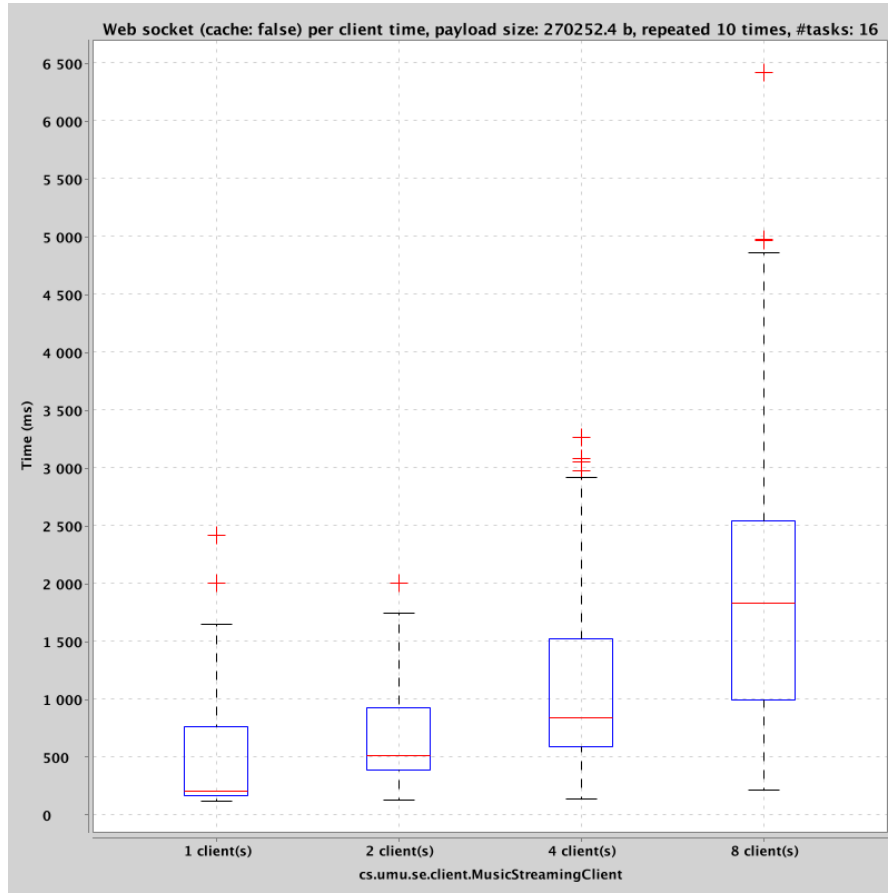


Figure 9: The result when measuring the per retrieve time when  $C$  clients are retrieving 5 % of a .mp3 file through a node's web socket with the caches in nodes turned off.

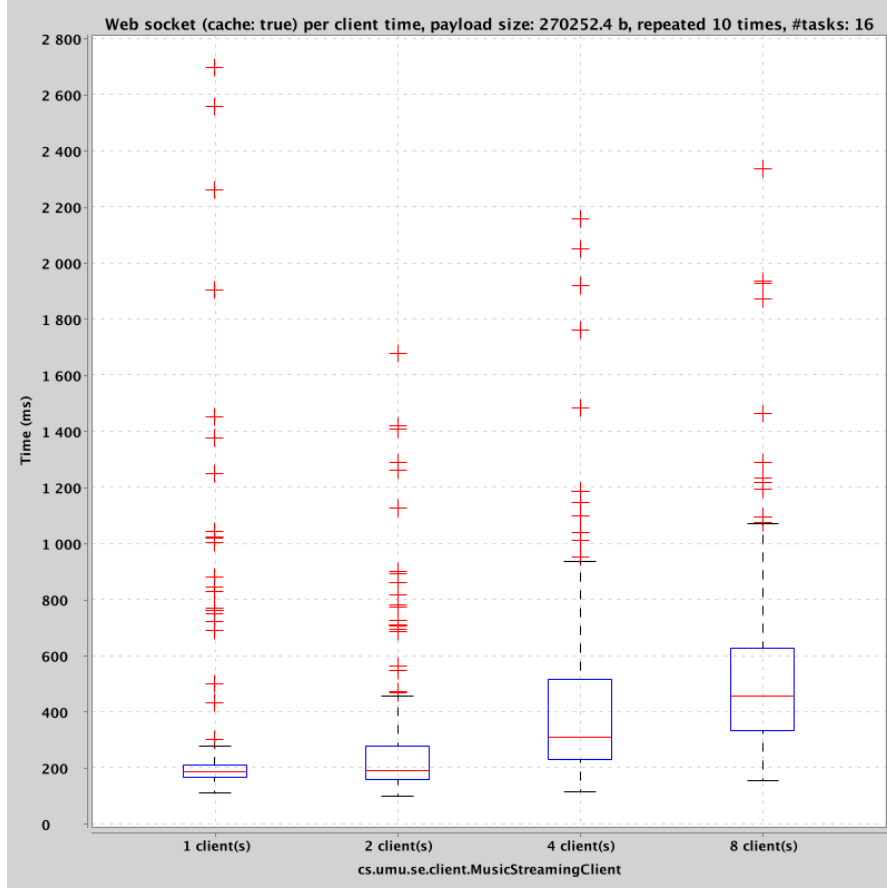


Figure 10: The result when measuring the per retrieve time when  $C$  clients are retrieving 5 % of a .mp3 file through a node's web socket with the caches in nodes turned on.

## 11 Conclusion

The results provided in Section 10 were compared to the theoretical values that were expected. When estimating the transfer time of a file Equation 1 was used where  $T$  is the transfer time in seconds,  $S$  is the size of the file in bits and  $N$  is the network speed in bits/s.

$$T(s) = \frac{S(bits)}{N(bits/s)} \quad (1)$$

As stated in Section 10 the clients' shared bandwidth was known to be  $\approx 70$  Mbit/s and the size of the files was also known. Hence the project group were able

to estimate the expected transfer time. Worth noting is that some overhead was expected which was estimated to be  $\approx 0.5 - 1$  seconds per transfer.

When looking at the results in Figure 3 and 4 we draw the following conclusions:

The transfer time of a file of 5 Mb was expected to take  $\frac{5405048 \cdot 8}{70 \cdot 10^6} \approx 0.62s$  plus  $\approx 0.5 - 1$  seconds of overhead. When performing 30 retrieves the lower theoretical time was set to  $\approx 33.5$  seconds and the upper time to  $\approx 48.5$  seconds. The median of the result with no cache seen in Figure 3 was  $\approx 34.7$  seconds which was within the interval. The corresponding results when using the cache seen in Figure 4 show a median of  $\approx 25.7$  seconds which was 23% below the theoretical lower bound. This fact indicates that the cache had a significant impact on the performance when retrieving files through gRPC between nodes.

The results in Figure 5 and 6 for measuring the total time it takes  $C$  clients to retrieve 16 .mp3 files gave the project group the following conclusions:

If we assume that every client of the total amount of clients  $C$  gets an equal share of the total available bandwidth we get that a client's bandwidth is  $\frac{70Mbit/s}{C}$ . By calculating the theoretical total transfer time when  $C = 8$  clients concurrently perform 2 sequential retrieves each ( $2 \cdot 8 = 16$ ) with no cache using Equation 1 we got  $\frac{5405048 \cdot 8}{(70 \cdot 10^6)/C} \approx 4.94s$  per retrieve. Adding the overhead of  $\approx 0.5 - 1$  seconds gave a  $bound_{lower}$  of  $(4.94 + 0.5) \cdot 2 \approx 10.8$  seconds and a  $bound_{upper}$  of  $(4.94 + 1) \cdot 2 \approx 11.9$  seconds per client to perform two retrieves in parallel with the other clients. The median value in Figure 5 was  $\approx 10.9$  seconds which was inside the interval. When looking at Figure 6 which used the cache the median is  $\approx 7.7$  seconds which is below the lower bound. Hence the project group concluded that the cache had a significant impact on throughput since it was a performance increase of  $\approx 29\%$ . Worth noting is that since we had limited bandwidth the network got congested really quickly, hence there was little to no gain in total time when jumping from 4 clients performing 16 retrieves together compared to 8 clients performing 16 retrieves together. We tried moving the nodes to run on more powerful machines but got the same results as shown in Figure 5 and 6. This led us to believe that the bandwidth was the bottleneck of the system. In an utopia where bandwidth is unlimited, we would have seen a linear decrease in the total retrieve time when doubling the number of clients until the node gets overloaded and becomes the new bottleneck. However, this is difficult for us to test since we have limited hardware to act as several concurrent clients and no way to drastically increase the bandwidth inside our test environment. We have provided some theoretical calculations of the limitations of our system in the end of this section.

The results in Figure 7 and 8 from measuring the per retrieve time when  $C$  clients

are retrieving a whole .mp3 file through a node's web socket made the project group draw the following conclusions:

We still assume the clients get an equal share of the bandwidth. We calculate an expected transfer time per request when  $C = 8$  to be  $\frac{5405048 \cdot 8}{(70 \cdot 10^6)/C} \approx 4.94s$  per retrieve. Adding the overhead of  $\approx 0.5 - 1$  seconds gave a  $bound_{lower}$  of  $(4.94 + 0.5) \approx 5.44$  seconds and a  $bound_{upper}$  of  $(4.94 + 1) \approx 5.94$ . Looking at the median in Figure 7 that did not use the cache we got  $\approx 5.1$  seconds. Looking at Figure 8 that did use the cache the median was  $\approx 3.9$  seconds which is 23% faster than not using the cache. Again, as stated above, we believe that the bandwidth is the limiting factor and when several clients concurrently tries to retrieve data on limited bandwidth the time per retrieve increases when the number of clients increase. If we had unlimited bandwidth the boxes in the figures would all be at the same horizontal level since the per client time would be the same regardless of how many clients that concurrently tries to retrieve files over the network. This would be true until we find the limit of what the node's hardware can handle, then the node becomes the new bottleneck.

The result seen in Figure 9 and 10 when measuring the per retrieve time when  $C$  clients are retrieving 5% of a .mp3 file through a nodes web socket gave the following conclusions:

This test was made to check if it was possible to buffer music, start listening and keep on buffering faster than the client consumes the music. We assumed that the whole file was 5 MB and that the buffer in the client was 5% of the file's total size. We also assumed that an average song is 180 seconds long and that 5% of an average file is 270253 bytes and 5% of 180 seconds is 9 seconds. Hence the per client retrieval time of 5% of a file must be faster than 9 seconds in order to let a client stream music without interruption.

We still assume the clients get an equal share of the bandwidth. We calculate an expected transfer time per request when  $C = 8$  to be  $\frac{270253 \cdot 8}{(70 \cdot 10^6)/C} \approx 0.25s$  per retrieve. Adding the overhead of  $\approx 0.5 - 1$  seconds gave a  $bound_{lower}$  of  $(0.25 + 0.5) \approx 0.75$  seconds and a  $bound_{upper}$  of  $(0.25 + 1) \approx 1.25$ . The median when not using the cache in Figure 9 was  $\approx 1.8$  and the median when using the cache was  $\approx 0.44$  seconds. Again, the cache seemed significant since the cached version was 75% faster. The reason for this extreme increase might be due to the fact the songs that were often requested on the non-caching test might have been stored in another node, hence the node needed to find the song in another node which took time. The test using the cache only had to do this once per requested song, therefore, saving a lot of time. Worth noting is that the outliers in Figure 10 match the times shown in Figure 9 which indicates that this was when the node needed to find files in other

nodes instead of its own local cache.

Regarding the theoretical limitations of *Spadgify* we made the following calculations:

We assumed that we had the hardware to simulate 100 concurrent clients, a bandwidth of 1 Gbit/s and that we were using three nodes running on different machines than the clients. We also assumed that each client got an equal share of the bandwidth according to  $\frac{1000Mbit/s}{100} = 10Mbit/s$ . In practice, a client needs to be able to fill its buffer faster than it consumes it in order to use the Spadgify music service without interruptions. We assumed that a .mp3 file is 5 MB and that the buffer in the client is 5% of the file's total size. We assumed that an average song is 180 seconds long and that 5% of an average file is 270253 bytes and 5% of 180 seconds is 9 seconds. Hence the per client retrieval time of 5% of a file must be faster than 9 seconds in order to let a client stream music without interruption. We also recognized that the time it takes for a node to fetch a file from another node increases logarithmically when increasing the number of nodes since the chord protocol is used. In this scenario, we keep the number of nodes set to three. With the above stated assumptions the following calculations were made:

$$T = \frac{270253 \cdot 8}{10Mbit/s} \approx 0.22s \quad (2)$$

Adding the overhead of 0.5–1 seconds to  $T$  we get  $bound_{lower}$  of  $(0.22+0.5) \approx 0.72$  seconds and a  $bound_{upper}$  of  $(0.22 + 1) \approx 1.22$  seconds. Both of these bounds are well within the 9 second hard limit stated above. By calculating the factor  $K = \lfloor \frac{9}{1.22} \rfloor = 7$  we can multiply our 100 clients with  $K$  to get the theoretical maximum number of clients concurrently being able to stream music from a node in the environment assumed above and still be within the 9 second threshold for filling a client's buffer. Note, this is assuming that the hardware of the nodes is not a limitation.

As stated in our project plan, we aimed to support 100 concurrent clients. Unfortunately, we don't have the hardware nor bandwidth to test this in practice but the mathematical calculations above seem to support our claim that our system would be able to handle such a load.

## References

- [1] Ion Stoica; Robert Morris, David Krager, M. Frans Kaashoek, Hari Balakrishnan. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*. <https://www8.cs.umu.se/kurser/5DV205/HT19/literature/stoica2001chord.pdf>. [Online; accessed 22-October-2024].
- [2] Wikipedia. *Spotify*. 2024. URL: <https://en.wikipedia.org/wiki/Spotify> (visited on 10/22/2024).
- [3] Wikipedia contributors. *Chord (peer-to-peer)* — *Wikipedia, The Free Encyclopedia*. [https://en.wikipedia.org/w/index.php?title=Chord\\_\(peer-to-peer\)&oldid=1166976065](https://en.wikipedia.org/w/index.php?title=Chord_(peer-to-peer)&oldid=1166976065). [Online; accessed 22-October-2024]. 2023.
- [4] Wikipedia contributors. *GRPC* — *Wikipedia, The Free Encyclopedia*. <https://en.wikipedia.org/w/index.php?title=GRPC&oldid=1236917462>. [Online; accessed 22-October-2024]. 2024.
- [5] Wikipedia contributors. *REST* — *Wikipedia, The Free Encyclopedia*. <https://en.wikipedia.org/w/index.php?title=REST&oldid=1250694921>. [Online; accessed 22-October-2024]. 2024.