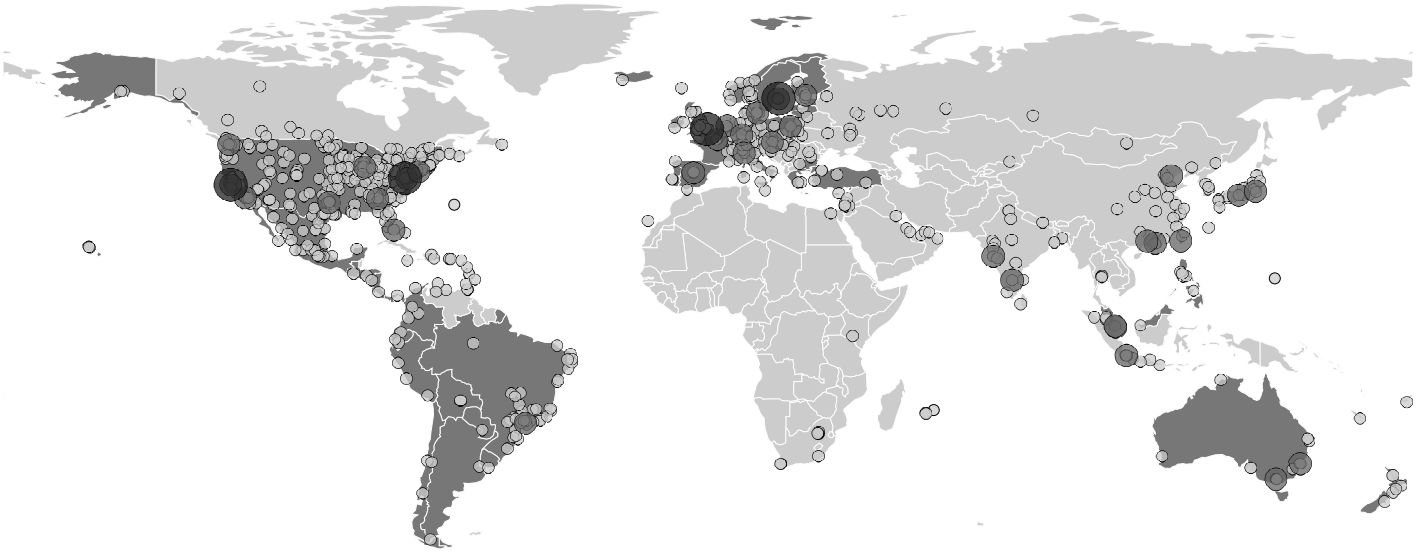
CHALMERS



A multi-CDN request routing strategy

Master of Science Thesis in Computer Science

OSCAR SÖDERLUND

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, October 2014

A multi-CDN request routing strategy

OSCAR SÖDERLUND

Examiner: PETER DAMASCHKE

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden October 2014

Abstract

In this thesis, we design and implement a multi-CDN request routing strategy for Spotify's audio files. Through A/B testing, we show that our strategy improves median download latency compared to Spotify's existing routing strategy.

Our strategy groups Spotify's users by autonomous system number and country, and uses a linear programming model on download latency log messages to generate routing weights on a group-by-group basis.

Our linear programming model generates routing weights with the goal of minimising request latencies, while also preserving a number of traffic volume constraints.

Acknowledgements

Thank you Niklas Gustavsson of Spotify, for suggesting this thesis, and for your invaluable guidance and support throughout the whole project.

Thank you Peter Damaschke of Chalmers University of Technology, for your theoretical advice and for guiding my report writing.

Table of Contents

4 Routing strategy

5 Implementation

6 A/B test

7 Results

8 Discussion

9 Appendix

Bibliography

1

Introduction

Background

Spotify streams music to over 40 million people every month.

Delivering a top notch streaming experience to every user is critically important. A big part of this is being able to stream music with no perceivable delay.

The technical term for delay is *latency*. Spotify denotes the time between a user initiating a stream and the start of music playback as *playback latency*. An important part of achieving low playback latency is being able to serve audio files at low *download latency*. Download latency is the time between requesting a file and receiving its initial contents.

Spotify streams audio files from their own data centres, but also from content delivery networks (CDN). One reason for why Spotify uses CDNs is to assure that every user can stream music from a nearby server, thereby achieving low download latency.

At the time of our thesis work, Spotify is still in the early stages of fully utilising CDNs. As such, there are many improvements to be made.

In this report, we propose one such improvement; a multi-CDN request routing strategy, with the goal of routing every user to the lowest latency CDN.

Goals

The goal of our thesis project has been to develop a routing strategy for Spotify's audio files, with the objective of minimising download latencies.

The important milestones in this project has been to:

- understand the problem domain
- find a suitable optimisation model that fits the problem domain
- produce an implementation of the routing strategy
- measure the performance of the routing strategy

Method

Location

We have carried out our thesis project at Spotify's office in Gothenburg, together with the team responsible for music streaming.

Domain and literature reviews

We have conducted a domain review on the subject of Spotify's music streaming infrastructure, with the goal of better understanding our problem and limitations on possible solutions.

We have also conducted a literature review, where we have looked at related work and a number of applicable optimisation algorithms.

Data retrieval and analysis

We have used latency log messages as input to our optimisation model.

Access to this data has been provided by Spotify, through their internal toolchains for log message analysis.

Iterative development

We have used an iterative workflow when developing our routing strategy.

For each iteration; we have implemented, tested and integrated the strategy into Spotify's music streaming infrastructure. Every development cycle has ended with a live test.

The end result presented in this report is the product of our final iteration of development and live testing.

A/B testing

We have used A/B testing to evaluate the effectiveness of our routing strategy.

A/B testing is a commonly used methodology, which is described in detail in the paper *Guide to controlled experiments on the web*[1]. Spotify's specific A/B testing methodology is described in chapter 2.

We have used Mann-Whitney's U-test to prove statistical significance in our A/B test results. The particulars of our A/B test is described in detail in chapter 6.

Limitations

We have limited our work to selecting and implementing one single optimisation model, as opposed to evaluating multiple optimisation models against each other. Lack of time is the major reason for this.

Any production code at Spotify, however experimental, must be thoroughly reviewed and tested. This has required us to spend a significant amount of work on iterative implementation; involving a rigorous process of code review, unit testing and integration testing.

This limitation has, in the end, enabled us to run live tests, the results of which are presented in chapter 7.

Scientific challenges

Significant domain-specific requirements

The main scientific challenge in our thesis work has been to develop a request routing strategy that fits Spotify's specific needs. Adapting a theoretical model to a set of practical requirements is always challenging, our case has been no exception.

For example, Spotify already has an existing system for routing requests that we are required to integrate with. In chapter 2, we learn precisely how Spotify's *storage-resolve* service works, and what kind of restrictions it imposes upon our routing strategy.

Unique problem domain

Our problem of routing requests to low latency CDNs may seem like a variation of the general load balancing problem, commonly encountered in distributed systems theory. This, however, is not the case.

The fundamental load balancing problem concerns distributing units of work across multiple machines. The concept of a single machine is not part of our domain.

We deal with distributing requests across multiple CDNs, which in turn are made up of many distributed clusters of machines. There is no notion of routing requests to specific machines. There is also no notion of how our routing affects the load of individual machines.

Inside a CDN, load is distributed among machines in a way that is opaque to us. Our problem is solely concerned with making optimal routing decisions to the entry points of these networks. How requests are routed once inside these networks is out of our control.

In chapter 2, we learn that our problem is rare, albeit not previously unheard of.

Local versus global optimisation

Another important aspect of our problem domain is dealing with global versus local optimisation.

We cannot blindly route every incoming request to the lowest latency CDN. We must also account for how our routing strategy affects overall latency. We must make sure not to trade slightly higher performance in the present for significantly lower performance in the future.

For example, in chapter 2, we learn about traffic volume commitments that our routing strategy absolutely must adhere to. We also learn about how CDN caches work, and how local optimisation may result in cold caches and future performance degradation.

Ethical aspects

Download latency not only affects the quality of experience for Spotify's users, it also affects the energy consumption in mobile devices. Every time someone uses Spotify to stream music through a smartphone or tablet, that device must divert extra power to its radio transmitter, increasing energy consumption. Spotify has an environmental responsibility to minimise their impact on energy consumption.

Ericsson has published a set of guidelines for smartphone app developers[2]. They strongly suggest limiting network usage, due to the high energy requirements of network transmitters in mobile devices. A network transmitter can be in several different modes, and prolonged data transfers requires the device to be in the most energy consuming mode.

Akamai, a global CDN provider, has found that achieving low latency is crucial to also achieving high data throughput[3]. Thus, by enabling downloads to start faster, we also shorten their duration. This leads to a reduction of energy usage for those who use Spotify to stream music through their smartphones and tablets.

Thesis outline

Chapter 2 summarises our domain review. Chapter 3 summarises our literature review, and presents the basics of linear programming. Linear programming is the foundation of our routing strategy, presented in chapter 4. In chapter 5, we present the implementation of this routing strategy.

Finally, chapter 6 describes how we have conducted our live tests, followed by test results in chapter 7 and our conclusions in chapter 8.

2

Domain review

This chapter contains background information on our problem domain.

We start with general aspects of Spotify, their clients and their backend, and then move on to content delivery networks, finally ending with some background on autonomous systems and Internet topology.

Music streaming at Spotify

Every instance of a song being played on Spotify is referred to as a *stream*. An essential part of the Spotify user experience is having all streams start with no perceivable delay.

For example, when playing music through Spotify's mobile app, you would expect music to start playing the moment you tap on a song. A noticeable delay, and the app would be experienced as sluggish. Ideally, streaming music through Spotify should be indistinguishable from playing MP3 files off the local hard drive.

This means that audio file downloads must be handled very swiftly. In order for a stream to start, the initial part of the audio file must first be downloaded. Spotify uses intelligent caching and prefetching strategies to make sure that most audio files are available even before a song starts playing.

However, caching and prefetching is not a universal solution, some streams can never be prefetched. These streams belong to the category of interactive streams.

Interactive streams

Streams either start as a result of a user action, such as clicking or tapping on a specific song, or as a result of playback progressing automatically from one song to the next in the currently playing context, such as a playlist. A stream initiated by a user action is referred to as an *interactive stream*.

Interactive streams pose a significant challenge, since user actions are stochastic and cannot be reliably predicted. This limits the efficiency of caching and prefetching strategies. Some interactive streams are bound to be uncached. In order for these streams to start no

perceivable delay, the Spotify client must be able to download the initial part of audio files with no perceivable delay.

Download latency

The time between initiating a file download and receiving the initial set of bytes for that file is formally referred to as *download latency*, or just latency.

Latency is generally measured in milliseconds (ms). Spotify has found that music playback delays larger than 250 ms become clearly noticeable to the user. In terms of download latency, 250 ms is not much time.

As an example, according to regular latency measurements performed by Verizon[4], the average latency of any request from Singapore to the US is around 170 ms, and that only applies to requests within Verizon's own network. Requests originating from other networks are likely to experience even higher latencies.

According to the same data, the average latency from Malaysia to the UK is just above 250 ms. Spotify has a data centre in London. Were Spotify to stream music from this data centre to users in Malaysia, those users would unavoidably receive a sluggish user experience. As such, it is imperative that Spotify route audio file requests to low-latency audio file storages.

Audio files

Before we look at how Spotify stores and delivers their audio files, it is imperative to understand that not all audio files are equal. Some tracks are streamed way more often than others.

Long-tail popularity distribution

The tracks in Spotify's music catalogue exhibit a so-called *long-tail distribution*. This distribution is visualised in figure 2.1, which is the result of plotting the number of requests to Spotify's 50000 most streamed audio files during a day.

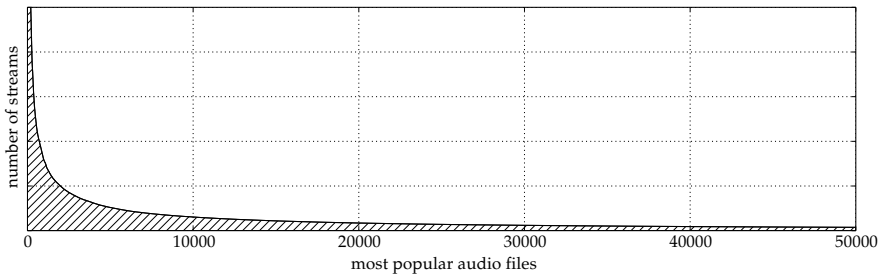


Figure 2.1: Popularity distribution of Spotify's 50000 most streamed audio files.

Figure 2.2 shows another aspect of Spotify's long-tail content distribution; a relatively small number of audio files (50000) account for over 50% of all streams during a day.

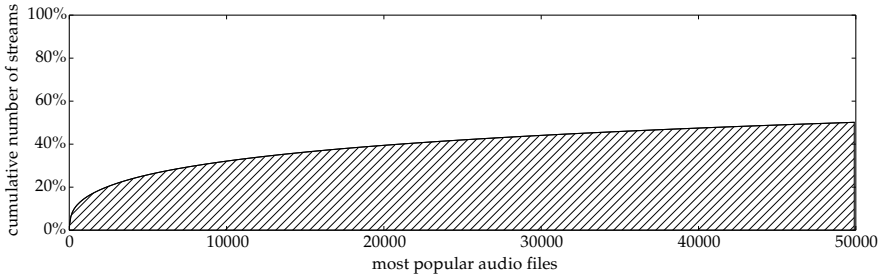


Figure 2.2: Cumulative number of streams for Spotify's 50000 most streamed audio files.

Audio file storages

Spotify serves their audio files from several sources; their own data centres, and a number of different content delivery networks.

Master storage

Spotify's *master storage* is hosted an Amazon S3. In the master storage, they store every file for every song in their entire music catalogue.

Amazon S3 provides high storage volumes at low cost, but at a trade-off of low access throughput. For this reason, Spotify serves audio files through additional storages, serving as high-throughput caches.

production-storage

Spotify's *production-storage* is a distributed content cache, hosted in their own data centres. Spotify has data centres in four locations: Stockholm, Sweden; London, England; Ashburn, USA and San Jose, USA.

Figure 2.3 highlights these locations on a map. The darker-shaded countries are markets where Spotify is officially launched, as of June 2014.

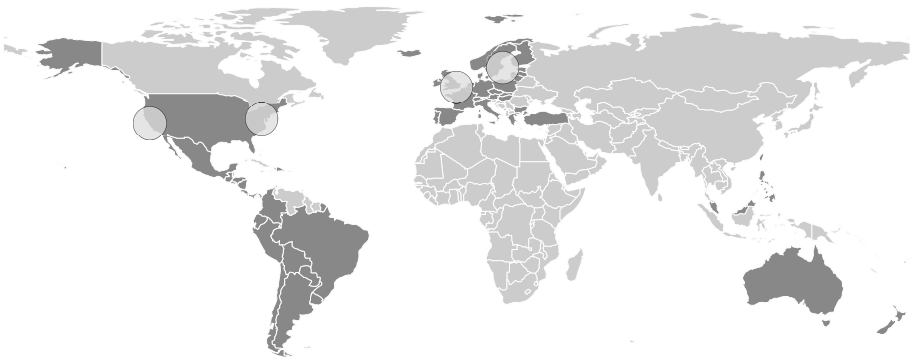


Figure 2.3: Deployment map of Spotify's production-storage.

The strength of Spotify's production-storage is that it has enough machines in every data centre to keep every frequently streamed audio file permanently in cache. However, it also has two major weaknesses; geographical reach and bandwidth capacity.

Spotify's data centres are located in North America and Europe. As such, South America and Asia, where Spotify is also available, will inevitably experience higher latency. This latency is acceptable for many of their backend services, but it is not acceptable for music streaming.

Moreover, the audio file traffic from the production-storage has to travel across the same network links that connect the Spotify's clients with their backend services. As Spotify has grown to over 40 million active users, in order for their data centre bandwidth to suffice for their backend services, the bulk of audio files must be served from elsewhere.

As such, to complement the weaknesses of their production-storage, Spotify also streams music through a number of external content delivery networks.

Content delivery networks

A *Content delivery network* (CDN) provides a distributed network of cache servers, which their customers can redirect traffic to. The CDN makes sure that that every request is served by the nearest available cache server, for example through clever DNS configuration.

When a cache miss occurs in a CDN server, the CDN server will request the file from the customer's origin server, cache it and then serve the request. In Spotify's case, the origin server is the Master storage on Amazon S3. A cache miss is generally an expensive affair, incurring a significant latency penalty.

CDN providers offer different payment models. A common model is paying a predetermined cost per gigabyte served. This per-gigabyte price is commonly negotiated through committing to pay for at least a predetermined amount of gigabytes per month. A higher traffic volume commitment can help negotiate a lower per-gigabyte price. Spotify makes

traffic volume commitments to their CDN providers, which significantly impacts our routing strategy.

All in all, a CDN works similarly to Spotify's production-storage, but with different strengths and weaknesses.

Differences between external CDNs and Spotify's production-storage

The most important difference between CDNs and Spotify's production-storage is that Spotify must compete with other content providers for cache space in external CDN servers. If an audio file is not requested often enough, it will get evicted from the CDN caches to the benefit of more frequently requested content, possibly from other content providers than Spotify.

As such, only the most frequently streamed part of Spotify's catalogue is viable for streaming via CDN. However, as shown in figure 2.2, this part of the catalogue still comprises the majority of all streams.

Different CDN providers employ different strategies in their hardware deployment, which also gives them different strengths and weaknesses.

One strategy, employed by CDN providers such as Level 3, EdgeCast and Amazon CloudFront, is to strategically select a small number of important locations along the Internet backbone, and serve requests with significant cache and performance capabilities from these few locations.

Figure 2.4 shows a map of EdgeCast's hardware deployment. On their website, they describe their strategy as building what they call *SuperPOPs*, points of presence at *"a small number of strategic global locations near Internet Exchange Points"* [5].

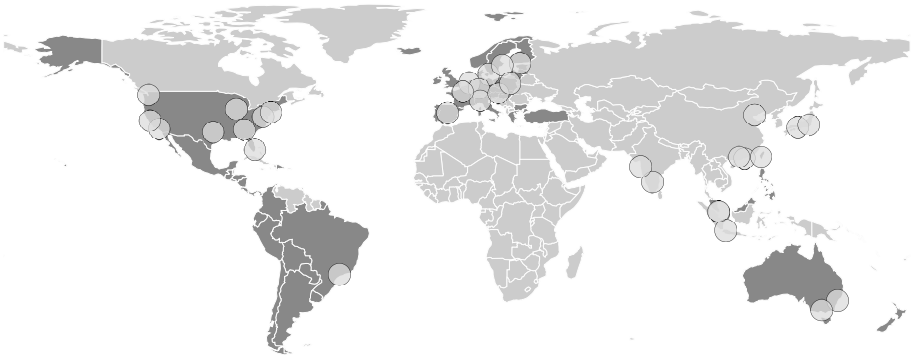


Figure 2.4: Deployment map of Edgecast's Super-POPs.

A different strategy, employed most prominently by the CDN provider Akamai, is to maintain a highly distributed deployment, which in turn is supported by a number of higher-tier cache layers. Akamai refers to the machines in their outermost cache layer as *edge nodes*, and their higher-layer caches as *midgress*.

Akamai argues that the way to consistently deliver high performance is to have servers close to every potential user [6]. The latency measurements from Verizon confirm this notion, geographical distance to the nearest server is highly impactful on latency.

Figure 2.5 shows a map of Akamai's edge nodes.

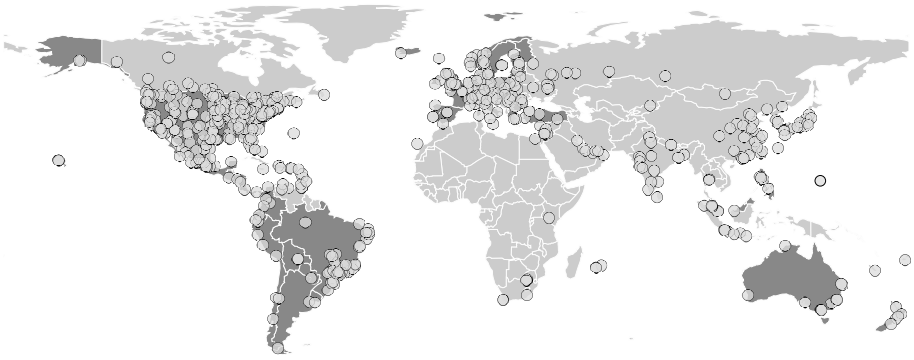


Figure 2.5: Deployment map of Akamai's edge nodes.

The differences between using distributed and centralised CDNs become apparent when considering how their caching mechanisms affect Spotify's content.

CDNs with centralised deployments will be able to provide more raw cache capacity, incurring fewer cache misses, while only being available at a limited amount of locations.

CDNs with highly distributed deployments will be available closer to the user, in theory providing lower latency. However, the size of Spotify's music catalogue, combined with the limited cache size of smaller edge node machines, increases the probability of cache misses. Akamai does account for this with their midgress layer, but even a cache miss in the edge nodes followed a cache hit in the midgress could incur a noticeable latency penalty.

To sum up, different CDNs have vastly different strategies in their deployments. Most likely, each CDN, including Spotify's production-storage, has its own strengths and weaknesses. One storage might perform well for some subset of users, while performing worse for another subset of users.

In our routing strategy, described in chapter 4, we partition our users into such subsets, and attempt to route the majority each subset to the highest performing storage.

A/B testing at Spotify

A/B testing is a technique for conducting and measuring the outcome of controlled experiments. This methodology has grown popular among data-driven technology companies, such as Spotify[7].

Being data-driven means basing important decisions on evidential data, as opposed to intuition or opinion. This significantly affects how new features at Spotify are developed and released.

At any given point in time, Spotify will be A/B testing a high number of new features and tweaks.

In our work, we have used Spotify's A/B testing methodology to evaluate the effectiveness of our routing strategy. We have also used tools from Spotify's A/B testing framework to implement our routing algorithms.

In this section; we explain Spotify's A/B testing methodology and the purpose of their internal A/B testing framework.

Overview

The purpose of an A/B test at Spotify is to compare the performance of multiple different variants of some feature. The Spotify users are divided into separate groups, each group being subjected to different variants of the feature being tested.

The group of users subjected to the new variant of a feature is commonly referred to as the treatment group, while the group of users subjected to the old variant is referred to as the control group.

The outcome of the A/B test is decided by measuring some test statistic for the treatment and control groups. At Spotify, this metric could for example be the median length of listening sessions in minutes, or, as in our case, the median latency of audio file downloads.

The desired outcome of an A/B test is to prove a statistically significant difference in the test statistic between the test groups.

Defining groups

Spotify has built an internal A/B testing framework that enables them to easily partition their users into treatment and control groups.

The way this is achieved is by uniformly distributing all Spotify users among 1000 base groups, numbered from 0 to 999.

When creating a new A/B test, treatment and control groups are formed by partitioning these base groups.

For example; in a simple A/B test with two groups, group A could consist of base groups $[0, 499]$, while group B consists of base groups $[500, 999]$.

Independence across tests

At Spotify, any number of A/B tests will be running simultaneously. In order to assure independence across tests, their A/B testing framework requires that every test be given a

unique test name.

The test name, which is a string, is used as entropy when partitioning users into base groups. Users will thus belong to different base groups across different tests.

Independence is achieved by keeping a separate partition of all users, into 10000 different buckets. This partition is consistent across tests, it never changes.

1000 unique base groups are created by Fisher-Yates shuffling[8] the 10000 buckets, using a linear congruence generator[9] with the test name as entropy.

Significance testing

The outcome of an A/B test is generally the rejection or failure to reject some null hypothesis. A common null hypothesis is that there is no difference in the primary metric between the test groups.

Rejecting the null hypothesis is achieved through a statistical test, such as Student's t-test. The test results in a p-value, which signifies the probability that the null hypothesis is true, given samples from the treatment and control groups.

A p-value of 0.05 or lower is commonly interpreted as grounds to reject the null hypothesis.

When the sample data does not follow a normal distribution, other tests than Student's t-test are better suited. For example, in our own A/B test, we have used Mann-Whitney's U-test [10], which does not require the test data to follow a normal distribution.

Logging at Spotify

Spotify is a data-driven company, and the bulk of this data is derived from collecting and analysing application logs.

The Spotify clients log a lot of valuable information, such as performance metrics. At regular intervals, these log messages are published to the backend for storage and further analysis.

The performance metrics from these log messages are for example used to calculate test statistics in A/B tests.

Topics and schemas

Spotify clients publish their log messages to different topics. Each topic has its own message schema and every schema contains one or more fields.

An example of a topic is the *Download* topic. A log message is posted to the Download topic whenever a client downloads an audio file. Some of the performance metrics this log message contains is initial latency to the storage from which the audio file was downloaded.

We use log messages from the Download topic as input to our optimisation model, described in chapter 4.

Storing log messages in HDFS

Every log message posted to every topic by every client eventually ends up in Spotify's HDFS cluster.

HDFS is specifically designed to integrate with Apache Hadoop, a popular Map-Reduce framework for distributed computation on large amounts of data.

Batch processing with Hadoop and Hive

Apache Hadoop [11] is the engine that powers many of today's data-driven companies. It enables complex computations on huge data sets.

Hadoop computations are referred to as jobs. Jobs are programmed in Java and submitted to run in a Hadoop cluster.

Programming and debugging Hadoop jobs can be time consuming. Many times, the computations to be performed are relatively simple, such as computing test statistics for an A/B test.

For the simple cases, Spotify uses Apache Hive [12], which compiles SQL queries to Hadoop jobs.

Download latencies

Through a Hive-query on log messages from the Download topic, we have investigated the frequency distribution of download latencies.

A histogram of 24 hours worth of logged download latencies is shown in figure 2.6.

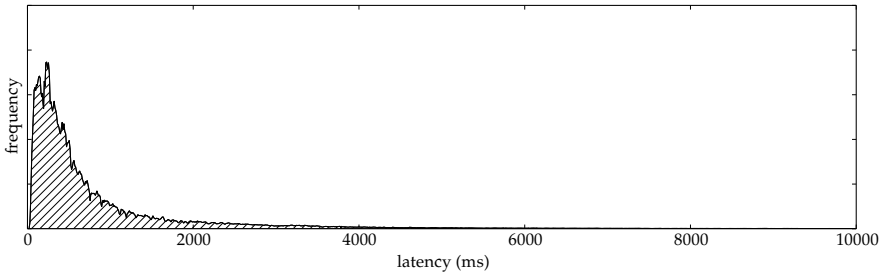


Figure 2.6: Distribution of logged download latencies.

We see that, similarly to the popularity distribution of audio files, the frequency of logged download latencies is a long-tail distribution.

The plot in figure 2.6 is cut off at 10000 milliseconds, but logged latencies up to the order of 100 seconds exist. These extreme latencies could possibly be the result of downloads stalling due to lost Internet connections.

The point to be made about download latencies is that their distribution contains a significant amount of high-magnitude outliers. This makes the sample mean of logged download latencies highly unreliable, since it is sensitive to these many outliers.

This becomes important to our routing strategy, defined in chapter 4, since we calculate expected values of latency distributions for subsets of Spotify users.

Spotify's regions

Spotify internally partitions countries into a number of different regions. These are *EU 1* to *EU 4*, *NA*, *LatAm* and *APAC*.

Figure 2.7 show these regions on a map, highlighted with a darker shade.

These regions become important to our routing strategy in chapter 4, when we define per-region traffic volume constraints in our optimisation model.

An important thing to note about these regions is that they only include countries where Spotify is officially launched.

(a) NA

(b) LatAm

(c) EU 1

(d) EU 2

(e) EU 3

(f) EU 4

(g) APAC

Figure 2.7: Spotify's regions.

The storage-resolve service

Spotify has a backend service that is specifically responsible for routing incoming audio file requests to audio file storages. This service is called *storage-resolve*.

In chapter 4 and chapter 5, we describe how we have extended the storage-resolve service to implement our routing strategy.

In this section, we describe in greater detail how the service works and what its purpose is.

Overview

The purpose of the *storage-resolve* service is to route audio file requests to storages where they may be downloaded. As such, whenever a client initiates an uncached stream, it will make a backend request to the *storage-resolve* service.

Routing rules

The storage-resolve service is configured with a number of routing rules. These rules specifically dictate how incoming requests should be routed to storages.

An example of such a rule is a practice called *zero-rating*, where some specific ISPs offer unlimited traffic specifically for Spotify usage. The ISPs achieve this by whitelisting specific IP ranges. The *storage-resolve* service makes sure that every request originating from a user with a zero-rated ISP is routed to a storage with a whitelisted IP-address.

As another example is making sure that requests for long-tail content are always routed to production-storage, since these audio files are unlikely to be available in CDN caches.

The routing rules are manually defined in the service's configuration file. This configuration file is managed by a configuration management system called Puppet. This makes it easy for Spotify to publish new versions of the configuration file to every machine running the storage-resolve service.

Routing strategy

The routing outcome of every request is not specifically determined by any rule. In fact, the majority of requests can be routed to any audio file storage.

For these requests, Spotify has defined a specific rule, which divides all users into A/B test groups; one group per storage. The routing outcome for a request that is not rule-determined is determined by the A/B test group of the requesting user.

Spotify can control the amount of traffic each storage receives by changing the proportions of these per-storage A/B test groups. These proportions are defined in the service's configuration file.

This is Spotify's existing routing strategy. The weakness we observe in this strategy is that requests are randomly routed to storages, since the A/B test group any particular user will end up in is randomly determined.

We propose a strategy where these requests are instead routed to the storage with the lowest latency. We describe this strategy in chapter 4.

Autonomous systems

In our previous section on CDNs, we concluded that different CDNs are likely to perform differently for different groups of users. An important part of our work is to properly identify these groups.

A naive yet reasonable approach would be to group clients together by geographical proximity, since we know from Verizon's measurements that geographical distance is highly correlated with latency.

However, while geographical proximity may be correlated with latency, other factors, such as routing paths and network link capacity, are equally important to consider.

In our routing strategy, we group our users together both by their country, to capture the concept of geographical proximity, and by their autonomous system, to capture the concept of Internet proximity.

In this section, we explain briefly what autonomous systems are, and in what manner they affect latency.

Overview

The Internet is made up of a large number of smaller, interconnected networks. Such a smaller network is called an autonomous system (AS). Every AS is identified by a unique autonomous system number (ASN).

A detailed description of ASes, who operates them and how traffic flows between them can be found in the paper *"The Growing Complexity of Internet Interconnection"* [13]. The information in this section is taken from this paper.

Size and numbers

An AS is commonly operated by an ISP, corporation, university or government institution. The largest ASes are operated by ISPs, these are significantly larger than most other ASes, both in terms of number of Internet users and geographical spread.

According to a report on AS statistics from June 2014, the number of active ASes on the Internet is slightly above 47000 [14]. In 2010, it was reported by Akamai that the single largest of these ASes represented 6% of all Internet traffic[6].

Figure 2.8 shows the geographical reach of one of the largest ASes, Cogent, according to download latency log messages received by Spotify during 24 hours.

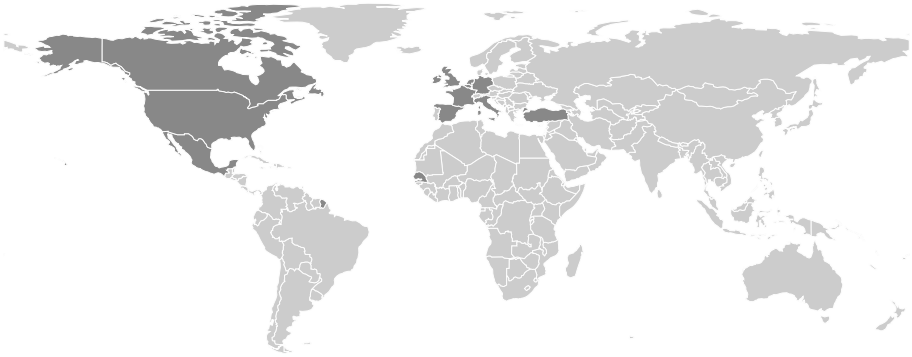


Figure 2.8: Map of Spotify requests from AS174, operated by Cogent.

We see that a large AS may span well over 10 different countries.

Figure 2.9 shows cumulative audio file requests received by Spotify during 24 hours, from the 500 most active ASes.

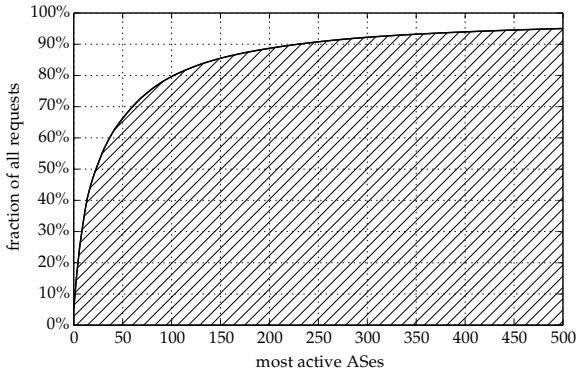


Figure 2.9: Cumulative request count of 500 most popular ASes.

We see that the 500 most active ASes comprise over 95% of all requests.

Impact on latency

An AS is a self-contained network. Within an AS, traffic is routed efficiently and latency is low. Across ASes, traffic has to cross special peering links. These peering links are generally few in number, prone to congestion and even occasional disconnection.

Peering links are an issue since they have to be established as the result of an agreement between two AS operators. In many cases, the benefits of maintaining a peering link are not mutual; operators of larger ASes charge operators of smaller ASes for traffic sent across peering links. As a result of this, operators tend to invest just enough resources into peering links to maintain an acceptable quality of service, but no more.

When network entities on separate ASes communicate with each other, a routing protocol called BGP (Border Gateway Protocol) decides how packets are routed across peering links [15].

The default mode of BGP is to send packets across the path that requires crossing the least peering links. However, BGP routers are controlled by AS operators, and they may influence this routing to suit their own needs.

As time has passed, peering agreements between AS operators have grown in complexity. Today, many AS operators will attempt to optimise their peering expenses through a practice called traffic shaping, where they take special care to route traffic through the peering links that incur the least cost.

All in all, the aforementioned intricacies of cross-AS routing give us reason to believe that grouping by AS will help us identify clients with similar CDN performance.

Summary

In this chapter, we have taken a closer look at the most important aspects of Spotify and Internet infrastructure to our routing strategy.

We will frequently refer back to the information in this chapter when explaining how our routing strategy works, and in motivating our design choices.

3

Literature review

In this chapter, we explore related work and introduce the theory of linear programming.

We motivate why linear programming is a good fit for modelling our problem, and run through the theoretical prerequisites to understanding our routing strategy.

Related work

IDNS

Intelligent Domain Name Server (IDNS) is a CDN routing strategy developed at AT&T Labs[16]. It can be used by CDNs wishing to route portions of incoming requests to other CDNs. This increases the ability of handling periods of particularly high load and sudden traffic spikes.

The routing strategy is based on measuring CDN performance and aggregating over over the IP address prefix of the requesting clients. The method of grouping clients by IP address prefix is originally described in another publication from AT&T Labs[17].

For every group of clients with the same prefix, they generate a probability-based routing table that includes every CDN, such that the highest performing CDN for every group handle the majority of requests from that group. They do not specify in detail which method they use to generate these probabilities.

The report includes test results from a live test where traffic is routed across two different CDNs. They conclude that their approach both increases overload protection and request latency.

WhyHigh

The paper *"Moving Beyond End-To-End Path Information to Optimize CDN Performance"* describes how Google uses a latency-based routing strategy for routing requests to edge nodes within their own CDN[18].

In this routing strategy, requests are consistently routed to the CDN edge node with the lowest measured latency, except for when that node is under high load. Latencies are measured and computed on a group-by-group basis. Clients are grouped by their IP address prefix.

The WhyHigh system, also described in the paper, is an internal diagnostics tool, which identifies groups of clients for which latency is highly inflated. The authors conclude that peering issues between autonomous systems is one prominent reason for inflated latencies, more so than geographical proximity.

Latency-based load balancing

Dynamic load balancing based on latency prediction is the title of Federico Piccinini's master thesis at Spotify from 2013[19].

In his report, Federico develops a routing strategy for load balancing requests between Spotify's internal access points. Spotify's access points are reverse proxies between Spotify's clients and the services in their backend. As such, every client request has to be proxied through an access point.

The purpose of Federico's routing strategy is to route every client to a low-latency access point.

Federico uses an adapted version of the ϕ-*accrual failure detector*, generating per-machine routing weights from per-machine mean and variance latencies, calculated from incoming log messages.

Summary

Our study of related work confirms that grouping clients together, most often by their IP address prefix, and generating per-group routing weights is a successful approach to optimising latencies.

However, it seems that in most cases where routing weights are used, the algorithms used to generate said weights do not seem to account for controlling traffic volumes.

We conclude that Spotify's wish to maintain tight control over traffic volumes, while also minimising latencies, makes any of the existing solutions unviable. We have instead decided to treat this as a novel problem.

Furthermore, since no related work go into detail in describing how their routing strategies work on an algorithmic level, we have decided that our course of action should be to identify an optimisation model that can capture Spotify's specific routing requirements, and devise a novel solution to the problem.

In conclusion, our greatest takeaway from our study of related work is that grouping clients together and using some sort of optimisation method to generate per-group routing weights seems to be a good starting point.

Finding an optimisation model

We have boiled down the knowledge from our domain review in chapter 2 into a summary of the most important requirements.

Requirements

- Our routing strategy should minimise latencies for Spotify's music streams.
- Our routing strategy must route every incoming request to one of the currently available CDNs.
- Our routing strategy must ensure that traffic volume commitments to individual CDNs are met.
- Our routing strategy must keep CDN caches from going stale, by ensuring that each CDN receives a minimum amount of traffic in multiple geographical regions.
- Our routing strategy must protect Spotify's production-storage, their own custom CDN, from excessive traffic volumes.

Outcome

Everything considered, our domain closely resembles a constrained optimisation problem. Constrained optimisation models focus on minimising some objective function, while preserving a number of constraint equations.

Although we have briefly considered other approaches, we have found no other approach offering the modelling power of describing constraints that constrained optimisation models offers.

In our case, we want to model the latencies of Spotify's music streams as an equation that can somehow be solved for how those streams should be routed. We additionally want to model our traffic volume requirements as constraint equations, such as to make sure that none of the requirements are violated.

We have found that linear programming models can sufficiently capture all of these requirements.

Linear programming

Linear programming is a fundamental subset of constrained optimisation, where every equation in the model is linear. It is a well-known optimisation method, with powerful solving tools readily available.

While occasionally used to model routing problems, linear programming is most prominently used to solve scheduling, manufacturing and transportation problems.

We base this short introductory section to linear programming on the theory and examples from the books *"Elementary linear programming with applications"*[20] and *"Operations Research: applications and algorithms"*[21].

Overview

A linear programming problem has two principal components: a linear objective function and a set of linear constraint equations.

The objective function contains a number of unknown decision variables. The constraint equations are inequalities that constrain the possible values of the decision variables.

The goal is to find an allowed assignment of values to the decision variables, such that the objective function is minimized or maximized.

Formal notation

A linear programming model is formally specified on the following form:

$$\begin{aligned}
\text{minimise} \quad & z = c_1x_1 + c_2x_2 + ... + c_nx_n && (3.1)\\
\text{subject to} \quad & a_{11}x_1 + ... + a_{1n}x_n \leq b_1 && (3.2)\\
& a_{21}x_1 + ... + a_{2n}x_n \leq b_2 && (3.3)\\
& ... && (3.4)\\
& a_{m1}x_1 + ... + a_{mn}x_n \leq b_m && (3.5)\\
\text{and} \quad & \forall i : x_i \geq 0 && (3.6)
\end{aligned}$$

The objective function (z) is a linear function of n unknown decision variables ($x_1 ... x_n$). $c_1 ... c_n$ are known coefficients (equation 4.1).

The constraint equations are a set of m linear inequality functions, containing the decisions variables.

$a_{11} ... a_{mn}$ and $b_1 ... b_m$ are known coefficients (equation 3.2 - 3.5).

Canonical and non-canonical form

The general linear programming model in equation 4.1 - 3.6 is specified on canonical form. This is equivalent to maximising the objective function, using the $\leq$ operator for all inequalities, and only allowing positive decision variables.

Conversely, a model on non-canonical form means that either the objective function should be minimised, the inequalities use the $\geq$ or $=$ operators, or the decision variables can be negative.

Every linear programming model on non-canonical form can be converted into an equivalent model on canonical form. The purpose of this conversion is to provide a uniform starting point for solution algorithms.

Since we use a toolkit (the GNU Linear Programming Toolkit, abbreviated as GLPK) to specify and solve our own linear programming model, we do not need to care about canonical form. Conversion to canonical form is handled automatically by the toolkit.

The simplex algorithm

A common solution algorithm for linear programming models is the simplex algorithm, which is a geometrical method that systematically explores the allowed solution space until an optimal solution is found.

GLPK provides an implementation of the simplex algorithm, which we use throughout our work, without any customisation or configuration. We refrain from going into the theory behind the simplex method, since this is not necessary to either understanding or implementing our routing strategy.

4

Routing strategy

In the previous chapter, we used a summary of our domain requirements to determine that linear programming is a suitable optimisation model for our routing strategy.

We used our studies of related work to determine that a good starting point for our own work is to group clients together and to generate per-group routing weights.

In this chapter, we turn these learnings into a formal routing strategy.

Overview

In short, our routing strategy consists of the following parts:

- A way of dividing Spotify's users into groups
- A way of generating per-group routing weights
- A strategy for what input data to base the routing weights on
- A strategy for when to update the routing weights
- A way of using the routing weights to route incoming music streams

We now drill deeper into these parts, explaining each one in detail.

Dividing Spotify's users into groups

We want to find some way of grouping Spotify's clients together, such that clients within the same group will share similar connectivity to the available CDNs.

We have seen several instances of related work where grouping clients by IP-address prefix has proven successful. However, we have also learned that this is a non-trivial task that requires up-to-date BGP routing table snapshots.

We group Spotify's clients by their autonomous system number (ASN). This means our groups are less granular than had we grouped by prefix, which may cause the grouping to be slightly less effective. However, this way of grouping is simpler, since IP address to ASN lookup tables are readily available.

Some autonomous systems are unproportionally large, spanning multiple countries and continents. For this reason, in order to make our groups slightly more granular, we add country of origin as an additional grouping criteria.

We thus create groups for every combination of ASN and country.

Generating per-group routing weights

We want our routing strategy to route each client to the lowest latency CDN. Grouping clients together makes our problem simpler, since we just need to find the lowest latency CDN for every group of clients, instead of for every individual client.

Simply routing every client within a group to the lowest latency CDN for that group is not sufficient. That does not give us control over traffic volumes. Instead we use routing weights, such that each CDN is given a weight for each group. For a particular group, a lower latency CDN is given a higher routing weight, and will thereby handle more requests.

We use a linear programming model to generate the routing weights. In this model, we describe latency as an objective function to be minimised and traffic volume requirements as constraint equations. The free variables of the model are the routing weights.

This way, we can guarantee traffic volume requirements, while simultaneously minimising latencies.

Latency log messages as input

We need data in order to figure out the connectivity from our groups to the CDNs. For this, we use download latency log messages from Spotify's clients.

Through these log messages, we know the latency of every recent audio file download and from which CDN the download was made. This information enables our linear programming model to generate routing weights that optimise download latencies.

Through these log messages, we also know the volume of requests from each group. This information enables our linear programming model to generate routing weights that do not violate traffic volume requirements.

Updating the routing weights

Since the connectivity on the Internet changes constantly, we update our routing weights at regular intervals.

An update consists of re-running our linear programming model on the most recent set of latency log messages, and obtaining a new set of routing weights.

We call the time between two updates a *time window*. Updates are performed at the edges between time windows.

The question remains of how many latency log messages to consider during each update. The most recent log messages are the most important, since they most accurately reflect the current state of connectivity on the Internet.

In our strategy, we include every log message from the most recent time window into our linear programming model. The resulting routing weights are active during the whole upcoming time window.

As such, our routing strategy finds an optimal set of routing weights based on latency data from recent, yet historic requests. We operate on the assumption that an optimal set of routing weights for the most recent time window will also be effective during the upcoming time window.

A simple illustration of time windows is found in figure 4.1. Here, Δ signifies the length of a time window, and T signifies the current time.

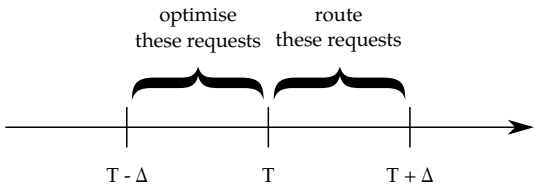


Figure 4.1: Time windowed optimisation scheme.

Using the routing weights

The final part of our routing strategy concerns how to actually use the per-group routing weights to route incoming music streams.

We have achieved this by extending Spotify's storage-resolve service, which is responsible for routing music streams to CDNs (chapter 2). We have added functionality such that it:

- computes which group the user belongs to
- looks up the current routing weight of every CDN for that group
- uses weighted random selection to determine the routing outcome

One detail worth mentioning is that we do not use pseudorandom numbers as entropy to the weighted random selection, since we want sequential requests from the same client to always be routed to the same CDN.

Instead, this entropy is produced through Spotify's A/B testing framework. We provide a more thorough description of this in chapter 5.

Linear programming model

Model definition

$$\text{minimise} \quad z = \sum_{\text{group}} \sum_{\text{CDN}} n_{\text{group}} \cdot w_{\substack{\text{CDN,}\\\text{group}}} \cdot \mathbb{E}\left[L_{\substack{\text{CDN,}\\\text{group}}}\right] \tag{4.1}$$

$$\text{subject to} \quad \forall\, \text{CDN} : \sum_{\text{group}} n_{\text{group}} \cdot w_{\substack{\text{CDN,}\\\text{group}}} \leq n \cdot c^{\,\text{max-traffic}}_{\text{CDN}} \tag{4.2}$$

$$\forall\, \text{CDN} : \sum_{\text{group}} n_{\text{group}} \cdot w_{\substack{\text{CDN,}\\\text{group}}} \geq n \cdot c^{\,\text{min-traffic}}_{\text{CDN}} \tag{4.3}$$

$$\forall_{\substack{\text{CDN,}\\\text{region}}} : \sum_{\substack{\text{group} \in\\\text{region}}} n_{\text{group}} \cdot w_{\substack{\text{CDN,}\\\text{group}}} \geq \sum_{\substack{\text{group} \in\\\text{region}}} n_{\text{group}} \cdot c^{\,\text{min-region-traffic}}_{\text{CDN,region}} \tag{4.4}$$

$$\forall\, \text{CDN} : w_{\substack{\text{CDN,}\\\text{group}}} \geq c^{\,\text{group-traffic}}_{\text{CDN}} \tag{4.5}$$

$$\forall\, \text{group} : \sum_{\text{CDN}} w_{\substack{\text{CDN,}\\\text{group}}} = 1.0 \tag{4.6}$$

$$\forall_{\substack{\text{CDN,}\\\text{group}}} : \text{group} \in G^{\text{default}} \Rightarrow w_{\substack{\text{CDN,}\\\text{group}}} = w^{\text{default}}_{\text{CDN}} \tag{4.7}$$

$$\text{and} \quad \forall_{\substack{\text{CDN,}\\\text{group}}} : w_{\substack{\text{CDN,}\\\text{group}}} \geq 0 \tag{4.8}$$

Decision variables

$w_{\substack{\text{CDN,}\\\text{group}}}$: routing weight of CDN for group

Input data

n	:	total number of log messages
n_{group}	:	number of log messages from group
$\mathbb{E}\left[L_{\substack{\text{CDN,}\\\text{group}}}\right]$	:	expected latency to CDN from group

Constraint parameters

$c^{\,\text{max-traffic}}_{\text{CDN}}$	:	maximum share of traffic to CDN globally
$c^{\,\text{min-traffic}}_{\text{CDN}}$	:	minimum share of traffic to CDN globally
$c^{\,\text{group-traffic}}_{\text{CDN}}$	:	minimum share of traffic to CDN from any group
$c^{\,\text{min-region-traffic}}_{\text{CDN,region}}$	:	minimum share of traffic to CDN from region
$G^{\,\text{default}}$	:	subset of groups predetermined to get default routing weights
$w^{\,\text{default}}_{\text{CDN}}$	:	default routing weight of CDN

Objective function

Our model is given as input the most recent latency log messages, each one containing a recorded latency from when some Spotify client downloaded an audio file from some CDN.

The objective function models the sum of the most recent logged latencies. This is the value we aim to minimise.

We already know the true value for the sum of latencies, but we want to infer what the sum of latencies would have been for varying sets of routing weights.

Through optimising the objective function, we aim to find the currently optimal set of routing weights, which we then use to route requests in the near future.

Routing weights

Through our latency log messages, we know the number of requests from each group, denoted in our model as n_{group}. The routing weight $w_{\substack{CDN,\\group}}$ determines the fraction of n_{group} requests that should be routed to a given CDN.

A routing weight is thus a real number between 0.0 and 1.0, and the routing weights within a particular group always sum to 1.0.

Expected latency

To reason about the expected value of the sum of latencies for varying sets of routing weights, we use the notion of *expected latency*.

We represent the latency of a request to a given CDN from a given group in our model as a random variable. We call this random variable $L_{\substack{CDN,\\group}}$. Every log message is considered a sample from one of these random variables.

Under a certain set of storage weights, the product $n_{group} \cdot w_{\substack{CDN,\\group}}$ represents the number of requests from a particular group routed to a particular CDN. It also represents the number of samples from $L_{\substack{CDN,\\group}}$ to include in the sum of latencies. $n_{group} \cdot w_{\substack{CDN,\\group}} \cdot \mathbb{E}\left[L_{\substack{CDN,\\group}}\right]$ represents the expected value of these samples.

Using the model's latency log messages, we estimate the expected value $\mathbb{E}\left[L_{\substack{CDN,\\group}}\right]$ for every combination of CDN and group.

The standard estimator of expected value would be the sample mean. However, recall from chapter 2 the long tail shape of latency distributions. These long tails contain a small but significant amount of high-magnitude outliers, rendering the sample mean unstable and prone to excessive variation.

For this reason, instead of the sample mean, we use the sample median as estimator of expected latency, since the sample median is not as susceptible to variation in the presence of high-magnitude outliers.

Constraints

As we learned in chapter 2, it is crucial that we abide by a number of traffic volume requirements. These requirements are captured by our model constraints.

By the term traffic volume, we mean the amount of individual requests routed to some CDN. Our domain requires us to manage traffic volume on a number of levels; globally, per-region and per-group.

On a global level, we must make sure that all traffic volume commitments to the CDN providers are met. We capture this in our $c_{CDN}^{\text{min-traffic}}$ constraint parameters. These parameters, defined for every individual CDN, express the minimum amount of global traffic that each individual CDN must receive. As such, these parameters (and all other constraint parameters related to traffic volume) are real numbers between 0.0 and 1.0.

Analogously to the minimum global traffic volume constraints, we use our constraints on maximum global traffic volume, $c_{CDN}^{\text{max-traffic}}$, to make sure that Spotify's own CDN does not receive excessive amounts of traffic.

On a per-region level, we must make sure that every CDN receives enough requests to keep the caches from going cold. We learned in chapter 2 that Spotify's partition of clients into regions is based on their country of origin. Since every group belongs to exactly one country, we can trivially infer how our routing weights affect per-region traffic volumes. The minimum per-region traffic volume constraints are captured in our $c_{CDN}^{\text{min-region}}$ parameters.

On a per-group level, we must make sure that we always keep receiving logged latencies to every CDN, even though some CDNs may not currently perform well. Without future measurements, we would not be able to detect when previously inferior CDN improves. We capture this in our $c_{CDN}^{\text{min-group}}$ parameters.

Default weights

G^{default} is a subset of groups for which we do not let the linear programming model generate routing weights. Instead, these groups receive a set of default per-CDN routing weights, w_{CDN}^{default}.

Many groups generate too few requests for us to reliably estimate the expected latencies. We use a default set of routing weights to route requests from all these groups.

For other groups, the differences in expected latency are small or insignificant. For these groups, the lowest latency CDN might change across samples, without any change in the underlying conditions to match. Letting the linear programming model generate routing weights for these groups might do more harm than good, since the weights might then be subject to frequent oscillation. For these groups, we also use our default set of routing weights.

Preprocessing

We transform latency log messages to input for our linear programming model through a preprocessing step.

Here, we determine which groups should be included in $G^{default}$ and receive default routing weights. This is decided by two filters, a request count filter and a minimum latency difference filter.

During preprocessing, we also calculate the expected latencies $\mathbb{E}\left[L_{\substack{CDN, \\ group}}\right]$.

Expected latency calculation

To calculate the expected latencies, we partition all log messages into a set of samples for every combination of group and CDN. We then take the expected latency for a particular group and CDN to be the sample median of the sample set for that group and CDN.

Request count filter

To filter out groups without enough logged latencies, we use a request count filter parametrised by α, a minimum amount of requests every storage must receive per group.

We add every group to $G^{default}$ where at least one storage has less than α logged download latencies.

Due to the uneven distribution of traffic across ASes, we expect only a small fraction of all groups to pass this filter. In chapter 2, we established that Spotify receives requests from around 16000 different ASes, and that the 500 most active ones account for 95% of all the traffic.

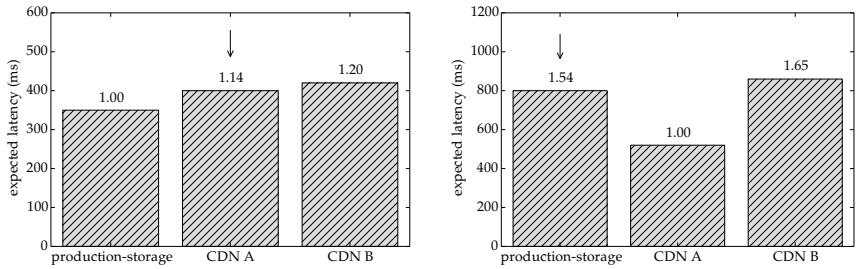
As such, we expect that even if our request count filter removes 15500 of 16000 groups, we will still be routing 95% of all incoming requests using our optimised storage weights.

Latency difference filter

To filter out groups with insignificant difference in expected latencies, we use a latency difference filter parametrised by γ, a minimum relative difference between the two lowest latency CDNs within a group.

γ is thus a real number greater than 1.0. $\gamma = 1.20$ means that all groups where the latency difference between the two lowest latency CDNs is less than 20% will be filtered out and given default weights.

For example, given expected latencies of two groups in figure 4.2, and a value of $\gamma = 1.20$; the group in figure 4.2a would be filtered out, while the group in figure 4.2b would be included.



(a) Excluded under $\gamma = 1.20$.

(b) Included under $\gamma = 1.20$.

Figure 4.2: Expected latencies from two example groups.

The number of groups filtered out under different values of γ is hard to predict, since the expected latencies vary over time.

We believe that the behaviour of our linear programming model makes the latency difference filter is essential, since to to minimise our linear objective function, a solver will give the highest weight possible to the storage with the lowest in the majority of groups.

For groups with high differences in expected latency, these extreme weight differences are desirable. For groups with low differences in expected latencies, this may be counterproductive and even be harmful. By filtering out these groups, we avoid unnecessary and possibly counterproductive optimisation.

5

Implementation

In this chapter, we give some technical insight into how we have implemented our routing strategy, and what tools we have used.

Overview

Our routing strategy consists of two major components:

- a data processing pipeline starting with latency log messages and ending with routing weights
- an extension to Spotify's storage-resolve service enabling it to use routing weights

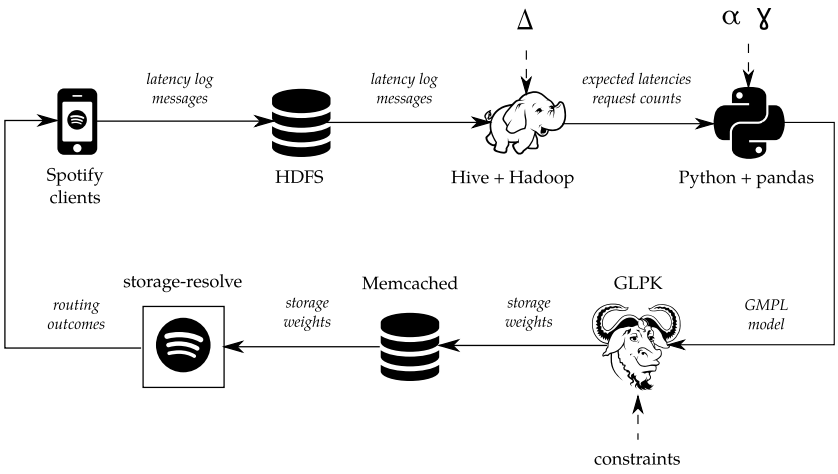


Figure 5.1: High-level overview of our routing strategy implementation.

Data pipeline

Our data processing pipeline consists of three parts. A Hive-job written in SQL, a preprocessing script written in Python, and a linear programming model written in GMPL (GNU MathProg).

Hive-job

Our Hive job transforms latency log messages from an interval in time into per-group and per-CDN expected latencies and request counts. It is written in SQL and compiles to a MapReduce job.

The Hive-job takes as input a starting date T, and a time window size Δ. It collects all latency log messages logged within the time window $\left[T - \Delta, T\right]$.

The output of the SQL-query is rows on the schema described in figure 5.2.

```
(group, CDN, region, request count, expected latency)
```

Figure 5.2: Schema of Hive query output.

Preprocessing step

Our preprocessing step filters out all groups which should receive fixed, default storage weights. It is implemented in Python using the NumPy[22] and pandas[23] libraries.

This step takes as input request count filter α, and a latency difference filter γ.

GMPL model

Once the output from Hive has been preprocessed, we interpolate the resulting data, together with our traffic constraints, into our GMPL model. The GMPL model is the implementation of our linear programming model described in chapter 4.

Our full GMPL model is included in the appendix in chapter 9.

We solve the resulting linear programming problem using the Simplex method included in the *glpsol* program in GLPK[24] (GNU Linear Programming Kit). We do not override any configuration parameters to the solver.

The solver calculates and outputs our routing weights. The formal output from the solver is a list of rows on the following format:

```
(group, CDN, routing weight)
```

Figure 5.3: Schema of GMPL model output.

Router

We use an extension of Spotify's storage-resolve service to route requests according to the routing weights generated by our linear programming model.

We use MaxMind's GeoIP[25] database to determine the ASN of an incoming request. This database maps IP ranges to autonomous systems. The country of origin is determined the same way.

Every instance of our extended storage-resolve service also runs a Memcached[26] daemon. When new routing weight have been generated, we use a remote Memcached client to put the routing weights into memory on all service instances. We associate the routing weights with the composite key formed by concatenating their corresponding ASN and country code.

Creating per-group and per-storage A/B test groups

Using Spotify's A/B testing framework, we have defined an A/B test specifically for weight-based routing. Each storage is assigned to an A/B test group, and the storage weights determine the sizes of the groups.

After having obtained the storage weights from Memcached, we look up which base group the client belongs to. We then determine which storage that base group is assigned to, and route the request to that storage.

Misc

As an example, consider figure 5.4. Here, we show how storage weights determine how a particular group of clients will be routed.

All clients in group g originate from the same country and ASN. There are three storages, $\{s_1, s_2, s_3\}$, each with an assigned weight $\{w_{g,s_1} = 0.1, w_{g,s_2} = 0.1, w_{g,s_3} = 0.8\}$.

Using the storage weights, we form an A/B test group for each storage from the 1000 base groups (see chapter 2 for a detailed introduction to Spotify's A/B testing framework). The sizes of the storages A/B test groups are proportional to their storage weights.

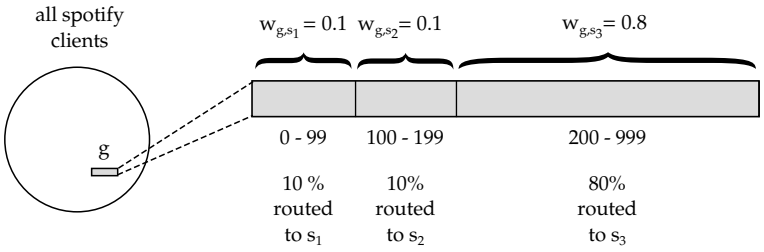


Figure 5.4: Example of weight-based routing within a group of clients.

A client from group g will with 10% probability belong to the A/B test group who gets routed to storage s_1, with 10% probability belong to the A/B test group who gets routed to storage s_2 and with 80% probability belong to the A/B test group who gets routed to storage s_3.

This means that the magnitudes of a group's storage weights proportionally determine the amount of traffic routed to each storage.

By using a linear programming model that assigns high storage weights to storages with low latency, we expect to reduce download latencies.

6

A/B test

We have conducted an A/B test with the purpose of comparing the effectiveness of our linear programming-based routing strategy to Spotify's existing routing strategy.

Test statistic

Our primary test statistic is the median of logged download latencies during the test.

Test duration

We have run our A/B test during exactly one day, a full 24 hours.

Test population

Our test population are all interactive streams handled by Spotify's San Jose data centre for the duration of the A/B test.

We have kindly been allowed to deploy our customised storage-resolve service in the San Jose data centre for the duration of the test. Since this is one of four data centres in total, we expect around 25% of all streams to be handled by this data centre.

Excluded requests

We have excluded all streams from zero-rated clients from the population. We have also excluded streams for long-tail music. The reason for excluding zero-rated clients is that they must always be routed to a whitelisted CDN. The reason for excluding long-tail music streams is that they must always be routed to Spotify's production-storage. See chapter 2 for a detailed discussion on zero-rating and long-tail music.

We have also excluded all requests for which the country or ASN of the requesting client can not be determined. This occurs when the IP of the requesting client is not present in MaxMind's GeoIP database.

Included requests

In order to determine the relative size of our test population, in relation to the total number of streams during the test, we have analysed the download log messages from a 24-hour period prior to the test.

Figure 6.1 shows the relative size of our test population during this 24-hour period, accounting for excluded streams, in comparison to the total number of streams (across all data centres).

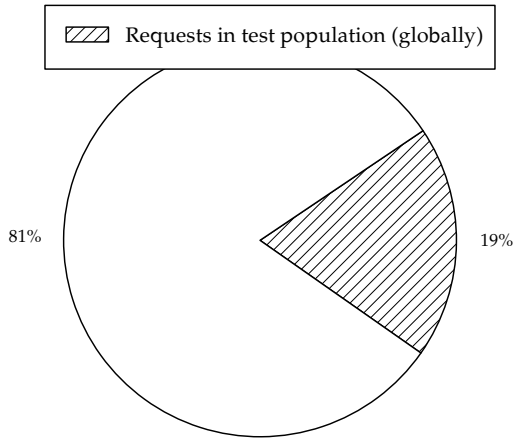


Figure 6.1: Pie chart showing amount of streams included in test population.

In total, our test population accounts for approximately 19% of Spotify's interactive streams during a 24-hour period.

Included countries

Figure 6.3 shows a geographical heat map of streams from our test population. A darker shade means more requests. The shades are assigned on a logarithmical scale.

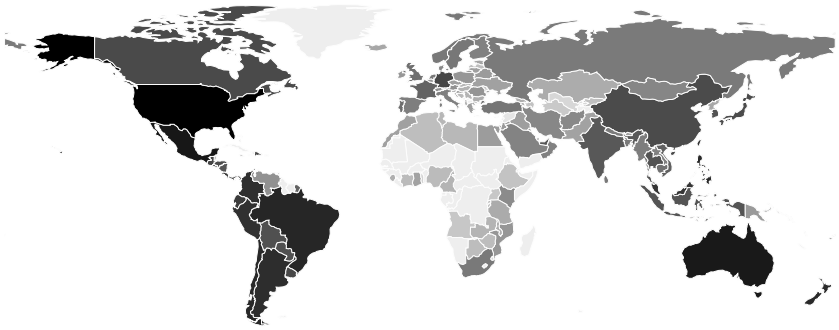


Figure 6.2: Heat map of per-country traffic from test population.

We see that our test population mainly consists of requests from the US, LatAm and APAC regions. More specifically, the vast majority of requests originate from USA, Mexico, Australia, and a number of South American countries.

Test groups

We have used Spotify's A/B test framework to partition the test population into two equally sized groups; a control group and a treatment group.

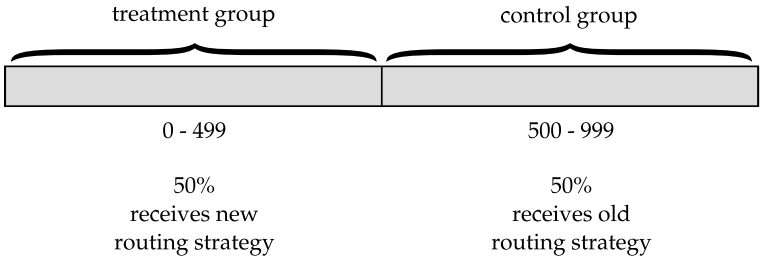


Figure 6.3: Treatment group and test group split of our test population.

Which test group an incoming request belongs to is determined by which base group the requesting user belongs to in our A/B test. Base groups 0 to 499 make up the treatment group and base groups 500 to 999 make up the control group.

Due to the independence of test groups in Spotify's A/B testing framework, we expect every group of ASN and country to contain an equal number of requests from our control and treatment groups.

Control group

Streams from the control group are exclusively routed by Spotify's existing routing strategy. This strategy is explained in detail in chapter 2.

Treatment group

Streams from the treatment group are routed according to the routing weights generated by our linear programming model. Our routing strategy is explained in detail in chapter 4.

Significance test

We use Mann-Whitney's U-test to determine if there is a statistically significant difference in the test statistic between the treatment and control groups.

We use Mann-Whitney's U-test since the test statistic is measured on a latency distribution. We know from chapter 2 that download latencies are prone to non-normal long-tail distributions.

This makes the more common Student's t-test inappropriate, since it is only suited to significance testing on data sets with a normal distribution.

Null hypothesis

Our null hypothesis is that the latency distributions of the treatment and control groups are identical.

By refuting the null hypothesis, we accept the alternate hypothesis that our treatment is effective.

We use the U-test to obtain a p-value for a double sided confidence interval that the null hypothesis is true, given 10000 randomly sampled latencies from both test groups.

If the p-value is less than 0.05, we refute the null-hypothesis and conclude that there is a statistically significant difference between the treatment and the control group.

We use the *mannwhitneyu* function from the SciPy statistics module to calculate the p-value.

Model parameters

Storages

During our test, Spotify was using 3 different audio file storages; their own production-storage and 2 different CDNs.

From here on out, we refer to the CDNs as *CDN A* and *CDN B*.

$\Delta = 1$ day

The time window size of our optimisation data has been 1 day.

The input to our optimisation model has thus been all download latencies logged by the test population during the 24 hours prior to our test.

$\alpha = 10000$ requests/storage

We have used a per-storage request count filter α with a value of 10000.

We have chosen this value by examining how different values of α affect the number of filtered out groups, and the amount of requests originating from these groups.

We have found that for $\Delta = 1$ day, a value of $\alpha = 10000$ makes $G^{default}$ account for less than 10% of total traffic.

$\gamma = 1.20$

We have used a latency difference filter γ with a value of 1.20.

For the input data, we have found this value to strike a balance between filtering out groups with low latency difference, while still generating optimised weights for a majority of the traffic.

Default groups

With $\alpha = 10000$ and $\gamma = 1.20$, we generate optimised weights for 101 different groups, less than 2% of the total amount of groups in the model. This means that $G^{default}$ contained the remaining 6074 groups.

As shown in figure 7.3, the optimised groups account for 81% of all requests. This is consistent with what we learned about AS traffic in chapter 2; that the top 500 ASes account for over 90% of Spotify's requests worldwide. The optimised groups are likely a subset of these 500 ASes.

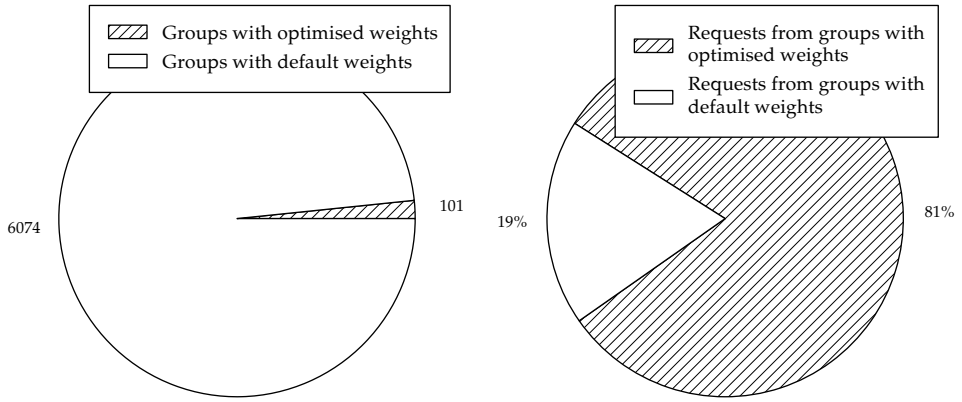


Figure 6.4: Pie chart of optimised groups vs. default groups.

Constraints

For the minimum group traffic per-storage constraint, we have required that every storage receives at least 10% of requests from every group (equation 6.1).

$$\forall \text{ CDN}: \ \mathsf{c}_{\text{CDN}}^{\text{group-traffic}} = 0.1 \tag{6.1}$$

Since Spotify has to compete for cache space in the external CDNs, we have required that every such CDN receives at least 20% of requests from every region (equation 6.2). Since our test population mainly consists of requests from LatAm, US and APAC, we have included only these regions in our constraints.

$$\forall \text{region} \in \{\text{LatAm}, \text{US}, \text{APAC}\}, \forall \text{ CDN} \in \{\text{CDN A}, \text{CDN B}\}: \ \mathsf{c}_{\text{CDN, region}}^{\text{min-region-traffic}} = 0.2 \tag{6.2}$$

For capacity reasons, we have required that Spotify's production-storage should receive at most 50% of requests globally.

$$\mathsf{c}_{\text{production-storage}}^{\text{max-traffic}} = 0.5 \tag{6.3}$$

Our default routing weights reflect how Spotify's existing routing strategy routed traffic at the time of the A/B test, which was to route 40% of requests to CDN A and CDN B respectively, with the remaining 20% being routed to production-storage (6.4).

$$w^{\text{default}}_{\text{production-storage}} = 0.2 \tag{6.4}$$

$$w^{\text{default}}_{\text{CDN A}} = 0.4 \tag{6.5}$$

$$w^{\text{default}}_{\text{CDN B}} = 0.4 \tag{6.6}$$

Routing weights

Figure 6.5 shows the routing weight output from our linear programming model as a scatter plot.

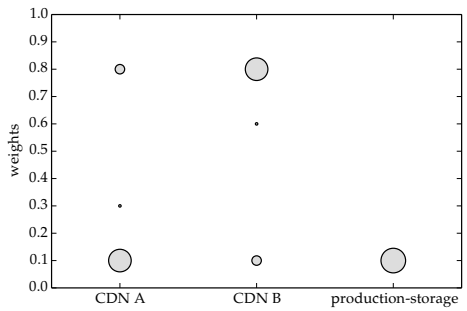


Figure 6.5: Scatter plot of routing weight values.

The x-axis indicates a CDN and the y-axis indicates weight. The point size indicates the amount of groups that received the routing weight indicated by the x and y-axis.

The scatter plot shows that CDN B received the highest possible weight in the majority of groups, while Spotify's production storage consistently received the lowest possible weight in every group.

A/A test

To complement the results from our A/B test, we have also performed an A/A test on the day preceding the A/B test.

In our A/A test, the treatment and control group are both routed using Spotify's existing routing strategy.

The test population during the A/A test is the same as during the A/B test.

The primary purpose of this test is to verify that the U-test works as intended on our latency distributions. Due to the amount of outliers, it might be the case that the U-test always gives a low p-value.

To show that this is not the case, we observe the results of the U-test on data sampled from a day where our A/B test is not active. We then know that the sampled data is taken from identical distributions.

The desired outcome of our A/A test is a p-value well above 0.05. If we also see a p-value well below 0.05 for our A/B test, we will be more certain that our outcome is not the result of random measurement variations.

7

Results

A/B test

Median latency

	Median latency
Control	242.5 ms
Treatment	233.0 ms

p-value

$$p = 0.016$$

A/A test

Median latency

	Median latency
Control	248.8 ms
Treatment	247.3 ms

p-value

$$p = 0.233$$

Analysis

Percentiles

Table 7.1 shows additional latency percentile values, for the control and treatment groups during the A/B test.

	Control	Treatment
5th	37.5	34.6
25th	116.2	108.1
50th	242.5	233.0
75th	542.7	532.8
95th	2225.8	2255.8

Table 7.1: Latency percentiles for control and treatment groups.

Countries

Figure 7.1 shows the per-country improvement in median latency for all countries in the test population containing at least 1 group (ASN and country) with optimised weights.

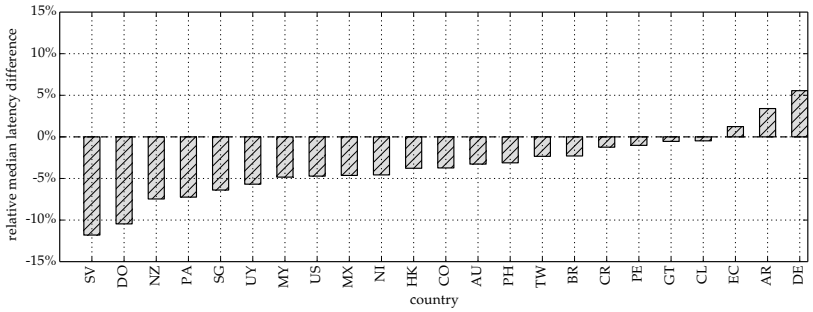


Figure 7.1: Latency improvement of countries containing groups with optimised weights.

Figure 7.2 shows a heat map of the data presented in figure 7.1. Darker shade means higher improvement.

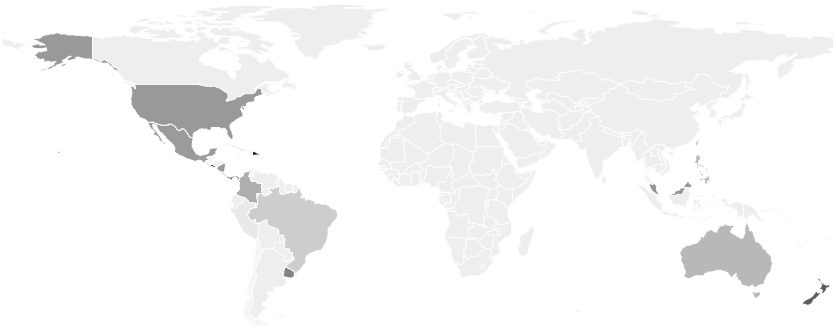


Figure 7.2: Heat map of per-country median latency improvement.

Higher latency difference filter

We suspect that higher differences in expected latency increases the likelihood that a group will benefit from our optimised routing.

To investigate this, we shrink the population by increasing γ to 1.50, meaning streams are only included into the population from groups where the latency difference between CDN is 50% or higher.

The median latency of the resulting population is presented in table 7.2.

	Median latency
Control	154.4 ms
Treatment	132.6 ms

Table 7.2: Median latency of groups with at least 50% expected latency difference.

Figure 7.3 shows the amount of groups and traffic included under the latency difference filter $\gamma = 1.50$.

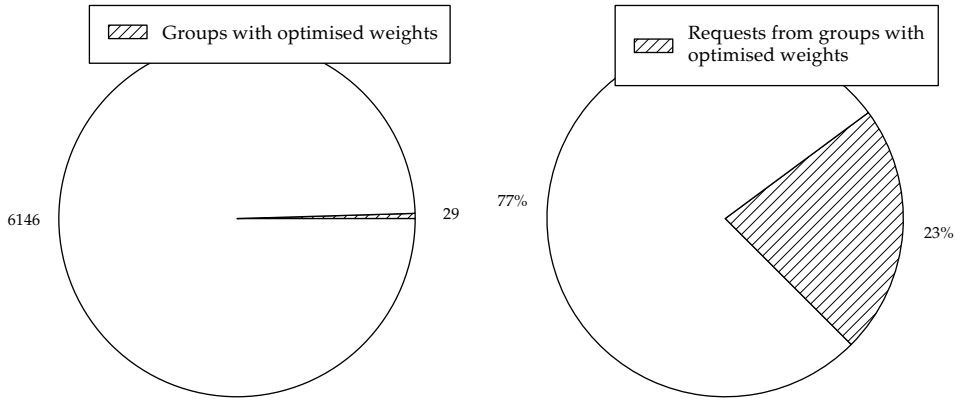


Figure 7.3: Pie chart of optimised groups vs. default groups for $\gamma = 1.50$.

We find that 23% of requests originate from groups where the difference in expected latency between storages is more than 50%.

For these requests, the measured improvement in median latency is approximately 11%.

8

Discussion

Conclusion

Successful A/B test outcome

From the results of our A/B test, we conclude that our routing strategy results in lower median download latency than Spotify's existing routing strategy.

The relative decrease in median latency between the control group and the treatment group is 9.5 milliseconds, or approximately 4%. Our U-test determines that this decrease is statistically significant.

With a p-value of 0.016, we refute the null hypothesis that the latency distributions of the treatment and control groups are identical.

Our A/A test reinforces our faith in the U-test, by giving a p-value of 0.233 for distributions that we know are identical.

We see decreases in all measured percentiles except the 95th percentile, where we see a 1.3% increase. We believe the reason that we fail to see an improvement in the 95th percentile is random noise in the long tail of the latency distributions. These latency measurements are likely the result of other phenomenons than ineffective routing.

Test statistic decrease mainly due to benefits in the US

Requests from the US make up the vast majority of requests in the test population. So much that the observed benefits in markets other than the US have no noticeable impact on the test population as a whole.

This can be seen in figure 7.1, where the latency improvement for the US, approximately 4%, is indeed the measured improvement of the entire population.

This is positive in the sense that our routing strategy is effective for the majority of incoming requests. This is important if the routing strategy is to be used to route every incoming request.

This is negative in the sense that by including requests from the US in the test population, we occlude more significant improvements in smaller markets.

Significant benefits in the Pacific and small markets

By looking more closely at subsets of the test population specific other countries, we see that the improvements are as high as 10% in El Salvador and the Dominican Republic. The relative amount of requests from these countries are however low enough to not affect the population as a whole.

The same conclusions apply to smaller countries in the Pacific, such as Malaysia and New Zealand.

We believe the increased effectiveness of our routing strategy in smaller markets to be closely tied to the deployments of CDN in these markets.

In large markets, such as the US, the CDNs are bound to have extensive deployments, and differences in performance are less likely.

In small markets, however, the nearest CDN edge node is more likely to be further away from the clients, both in terms of geographical distance and hops across peering links.

ASNs in these smaller markets are then more likely to have differences in expected latency to the various storages, which increases the effectiveness of our routing strategy.

Correlation between effectiveness and latency differences

Our measurements on the reduced population containing only groups where the latency difference is 50% or higher, we see that the test statistic improved by 21.8 milliseconds, or 11.4%.

Since these groups comprise 25% of the requests in the total population, we conclude that while the overall improvement in the test statistic is 4%, there are groups where the improvement is much more significant.

We believe these improvements to correlate with the difference in expected latency.

Extreme routing weights due to linear model

By looking at figure 6.5, which visualises the routing weights during our A/B test as a scatter plot, we see that our linear model results in what can only be referred to as “extreme” weight assignments.

By this we mean that across groups, the lowest latency storage will consistently be given the highest possible weight, except for some exceptional cases where the linear optimiser will the routing weights for a few groups to fulfil traffic volume constraints.

We see an example of this in figure 6.5, where the most common weight assignments for the three available storages are some combination of the weights $\{0.1, 0.1, 0.8\}$.

Only one single group diverges from this pattern, with a weight assignment of $\{0.3, 0.6, 0.1\}$. We see this as a clear example of the optimiser meeting traffic volume constraints.

However, the extreme weight assignments resulting from our linear model are not necessarily a bad thing.

Due to our latency difference filter, we only allow optimised weights for groups where the latency difference is big enough that extreme weights are warranted. The remaining groups, where extreme weights are not appropriate, are instead assigned a default set of weights.

Potential issues

Restrictive filters prevent improvements in small markets

We have seen that our routing strategy is likely to work well for groups with higher differences in expected latency. However, during our A/B test, we used a fairly restrictive request count filter.

This restrictive request count filter does not affect the test statistic as a whole, since these requests originate from the major ASes with high request counts.

It is however entirely possible that there are markets where these major ASes are not present. The groups in these markets would then be prevented from receiving optimised routing weights under restrictive request count filters, resulting in suboptimal routing.

Traffic patterns interfering with constraints

Upholding the traffic volume constraints from our linear programming model when routing future requests using the generated routing weights depends on the traffic patterns between time windows to be approximately similar.

For example, by generating routing weights using log messages from a Monday, we assume that the traffic patterns during the upcoming Tuesday, when the routing weights are used to route traffic, are similar enough that the traffic volume constraints will be upheld.

This is a problem since Spotify's traffic patterns are not only daily, but also weekly. People tend to stream more music on Fridays than Thursdays, which could cause problems when using optimised weights from a Thursday to route requests during a Friday.

In this case, it might be beneficial to use a weekly time window instead of a daily. This would account for varying, intra-week traffic patterns.

Minimum request count filter causing weight oscillation

In our A/B test, we have used a fairly restrictive minimum request count filter ($\alpha = 10000$ requests / storage), while also allowing the linear programming model to assign relatively low weights ($\forall s \in S : w_s^{\text{min-group}} = 0.10$).

Groups with low request counts, who just barely make the cut of having 10000 logged requests per storage, might not receive enough requests per storage during the upcoming time window, due to unbalanced routing weights.

Future work

Experiments with more permissive filter parameters

We believe more experiments are necessary to determine appropriate values for the request count filter. By allowing more groups with low request counts to receive optimised routing weights, we expect to see greater improvements in the test statistic in smaller markets.

Experiments on all traffic

We believe the traffic constraints to be a significant and powerful aspect of our routing strategy.

Through them, Spotify can use the existing per-region, and possibly in the future also per-country constraints to perform traffic shaping. That is, exercise granular control over traffic patterns.

One possible reason for Spotify performing traffic shaping would be meeting CDN traffic quotas. A common practice when dealing with CDN is to agree on paying for some minimum traffic quota. Using our routing strategy, Spotify could be very specific in deciding which traffic from which regions and countries should be allocated to meeting such traffic quotas.

The power of controlling traffic through our routing strategy is dependent on actually using the generated routing weights to route all incoming requests. During our A/B test, the linear programming model generated optimised weights based on 100% of the requests in the test population, but only 50% of those requests were subsequently routed using our routing strategy.

If our traffic constraints are to be 100% effective, the amount of requests routed by our strategy should be increased to 100%.

Publish routing weights as configuration file

The reason we used Memcached for storing weights in the implementation of our routing strategy is mainly to allow the weights to be frequently updated with low effort.

Having the storage-resolve service perform a Memcached lookup for every incoming request is however not desirable, since it increases the complexity of the service, and introduces a number of new failure modes.

A less complex way of implementing the weight-based routing would be for the service to look up the routing weights in a configuration file stored on disk. Since Spotify already uses Puppet to manage and publish their routing rules, we can imagine a similar solution for our routing weights.

Experiment with weekly optimisation

We believe that it would be very interesting to experiment with larger time windows for our routing strategy, for example by generating routing weights on a weekly basis as opposed to a daily basis.

Larger time windows means less effort in running the optimisation model and updating the routing weights.

The size of the time window is largely tied to our perception of how often we believe that expected latencies within groups are bound to significantly change.

9
Appendix

GMPL model

The following listing contains the implementation of our linear programming model in GNU MathProg.

The data definitions are to be provided as a separate file to the solver.

Listing 9.1: ”GMPL implementation of our linear programming model.”

```
# Copyright (c) 2014 Spotify AB

set STORAGES;
set REGIONS;
set GROUPS;
set GROUPS_TO_OPTIMIZE within GROUPS;
set GROUPS_WITH_DEFAULT_WEIGHTS within GROUPS;

param ExpectedLatencies{group in GROUPS, storage in STORAGES};
param RequestCounts{group in GROUPS, storage in STORAGES};

param GroupRegions{group in GROUPS}, symbolic;
param DefaultWeights{storage in STORAGES};

param MinGroupTraffic{storage in STORAGES} default 0.0;
param MaxGlobalTraffic{storage in STORAGES} default 1.0;
param MinRegionTraffic{region in REGIONS, storage in STORAGES} default 0.0;

var weights{group in GROUPS, storage in STORAGES}, >= 0, <= 1;

param GroupRequestCounts{group in GROUPS} :=
  sum{storage in STORAGES} RequestCounts[group, storage];

param GlobalRequestCount :=
  sum{group in GROUPS} GroupRequestCounts[group];

param RegionRequestCounts{region in REGIONS} :=
  sum{group in GROUPS} if (GroupRegions[group] == region)
                       then GroupRequestCounts[group]
                       else 0;

param MaxGlobalRequestCounts{storage in STORAGES} :=
```

```
  GlobalRequestCount * MaxGlobalTraffic[storage];

param MinRegionRequestCounts{region in REGIONS, storage in STORAGES} :=
  RegionRequestCounts[region] * MinRegionTraffic[region, storage];

minimize total_latency:
  sum{group in GROUPS, storage in STORAGES} (weights[group, storage] *
                                             GroupRequestCounts[group] *
                                             ExpectedLatencies[group, storage]);

subject to weight_sums{group in GROUPS}:
  sum{storage in STORAGES} weights[group, storage] = 1.0;

subject to min_group_traffic{group in GROUPS, storage in STORAGES}:
  weights[group, storage] >= MinGroupTraffic[storage];

subject to max_global_traffic{storage in STORAGES}:
  sum{group in GROUPS} (GroupRequestCounts[group] *
                        weights[group, storage])
  <= MaxGlobalRequestCounts[storage];

subject to min_region_traffic{region in REGIONS, storage in STORAGES}:
  sum{group in GROUPS} (if GroupRegions[group] == region
                        then GroupRequestCounts[group] *
                        weights[group, storage]
                        else 0)
  >= MinRegionRequestCounts[region, storage];

subject to default_weights{group in GROUPS_WITH_DEFAULT_WEIGHTS,
                           storage in STORAGES}:
  weights[group, storage] = DefaultWeights[storage];

solve;

for {group in GROUPS_TO_OPTIMIZE} {
  for {storage in STORAGES} {
    printf "%s\t%s\t%.2f\n", group, storage, weights[group, storage];
  }
}

end;
```

Constraints

These are the constraints we used for our model during the A/B test described in chapter 6.

Listing 9.2: "GMPL implementation of our linear programming model."

```
param DefaultWeights :=
CDN_A    0.40
CDN_B    0.40
storage  0.20;

param MinGroupTraffic :=
CDN_A    0.10
CDN_B    0.10
storage  0.10;

param MaxGlobalTraffic :=
storage  0.50;

param MinRegionTraffic :
       CDN_A     CDN_B :=
APAC    0.20      0.20
NA      0.20      0.20
LatAm   0.20      0.20;
```

Bibliography

1. R. Kohavi, R. M. Henne, D. Sommerfield*et al.*, in *Proceedings of the 13th aCM sIGKDD international conference on knowledge discovery and data mining*, (ACM, 2007), pp. 959–967.

2. Ericsson *et al.*, A smartphone app developer's guide (2014).

3. Akamai *et al.*, Highly distributed computing is key to quality on the hD web (2007).

4. Verizon *et al.*, IP latency statistics (2014) (available at http://www.verizonenterprise.com/about/network/latency).

5. Edgecast *et al.*, Edgecast - cDN locations (2014) (available at http://www.edgecast.com/network/map).

6. E. Nygren, R. K. Sitaraman, J. Sun*et al.*, The akamai network: a platform for high-performance internet applications, *ACM SIGOPS Operating Systems Review* **44**, 2–19 (2010).

7. H. Kniberg, A. Ivarsson *et al.*, Scaling agile@ spotify (2012).

8. R. A. Fisher, F. Yates, others*et al.*, Statistical tables for biological, agricultural and medical research., *Statistical tables for biological, agricultural and medical research.* (1949).

9. D. E. Knuth *et al.*, The art of computer programming, 3rd edn., vol. 2, *Seminumerical Algorithms* (1998).

10. H. B. Mann, D. R. Whitney, others*et al.*, On a test of whether one of two random variables is stochastically larger than the other, *The annals of mathematical statistics* **18**, 50–60 (1947).

11. T. A. S. Foundation *et al.*, Hadoop - open-source software for reliable, scalable, distributed computing. (2014) (available at http://hadoop.apache.org/).

12. A. Thusoo *et al.*, Hive: a warehousing solution over a map-reduce framework, *Proceedings of the VLDB Endowment* **2**, 1626–1629 (2009).

13. P. Faratin *et al.*, The growing complexity of internet interconnection., *Communications & Strategies* (2008).

14. T. Bates *et al.*, CIDR report (2014) (available at http://www.cidr-report.org/as2.0).

15. I. Van Beijnum *et al.*, *BGP: Building reliable networks with the border gateway protocol* (O'Reilly Media, Inc., 2002).

16. A. Biliris *et al.*, CDN brokering, *Computer Communications* **25**, 393–402 (2002).

17. B. Krishnamurthy, J. Wang *et al.*, in *ACM sIGCOMM computer communication review*, (ACM, 2000), vol. 30, pp. 97–110.

18. R. Krishnan *et al.*, in *Proceedings of the 9th aCM sIGCOMM conference on internet measurement conference*, (ACM, 2009), pp. 190–201.

19. F. Piccinini *et al.*, Dynamic load balancing based on latency prediction, (2013).

20. B. Kolman *et al.*, *Elementary linear programming with applications* (Gulf Professional Publishing, 1995).

21. W. L. Winston, J. B. Goldberg *et al.*, Operations research: applications and algorithms, (1994).

22. E. Jones, T. Oliphant, P. Peterson, others*et al.*, SciPy: Open source scientific tools for Python (2001–2001--) (available at http://www.scipy.org/).

23. W. McKinney *et al.*, in *Proceedings of the 9th python in science conference*, S. van der Walt, J. Millman, Eds. (2010), pp. 51–56.

24. GLPK *et al.*, GNU linear programming kit.

25. MaxMind *et al.*, GeoIP Organization (available at http://www.maxmind.com).

26. B. Fitzpatrick *et al.*, Distributed caching with memcached, *Linux journal* **2004**, 5 (2004).