Bachelor Informatica

Late Transmuxing: Improving caching in video streaming

Jelte Fennema

17th June 2015

Supervisor(s): Dirk Griffioen (Unified Streaming), Robert Belleman (UvA)

Signed: Robert Belleman (UvA)

Abstract

With the growth of the internet, usage has shifted from sending textual messages to streaming video. Lots of the different devices and players are used to watch these video streams. However, not all of those support the same streaming formats and that is why multiple different streaming formats are used to view the same video. Many known server setups already try to tackle the problem of serving these different formats fast and resource efficient. Most make use of proxy servers to reduce the load on the storage back end. Some use these proxy servers as a caching layer or a content delivery network (CDN), some others use them for on the fly conversion. None of the setups however, utilise the power of caching and on the fly conversion on the same server. Creating a setup that combines both, leads to a setup with before unseen capabilities. Videos can be served directly from the proxy server, in formats that have never been requested from the server before. Compared to other setups this lowers internal traffic and reduces load on the storage server. All of which results in faster, cheaper and more efficient video streaming.

Contents

1 Introduction **5**
1.1 Traditional setups . . . 6
1.2 Approach . . . 6

2 Problem analysis **7**
2.1 The single server setup . . . 7
2.2 The CDN setup . . . 8
2.3 The IsmProxyPass setup . . . 9
2.4 Combining the setups . . . 10

3 Design considerations **11**
3.1 HTTP servers . . . 11
3.1.1 General differences . . . 11
3.1.2 USP integration . . . 11
3.1.3 Caching . . . 12
3.1.4 Scripting . . . 12
3.1.5 The decision . . . 12
3.2 What responses to cache . . . 12
3.2.1 The decision . . . 13
3.3 Caching range requests . . . 13
3.3.1 Ignore range . . . 13
3.3.2 Match exact range . . . 13
3.3.3 Ranger . . . 13
3.3.4 The decision . . . 14

4 Implementation **15**
4.1 Initial implementation . . . 15
4.2 Final implementation . . . 17

5 Experiments **19**
5.1 Setup . . . 19
5.1.1 Virtual machines . . . 19
5.1.2 The different setups . . . 19
5.1.3 The tests . . . 19
5.2 Results . . . 21
5.2.1 Cold cache . . . 21
5.2.2 Cache filled with the same format . . . 25
5.2.3 Cache filled with another format . . . 29

6 Discussion **35**
6.1 Explanation of the results . . . 35
6.1.1 DASH and ISS versus HLS and HDS . . . 35
6.1.2 Internal traffic with a cold cache . . . 35
6.1.3 Cache usage with a cold cache . . . 36
6.1.4 Transfer speed and latency with a cold cache . . . 36
6.1.5 Results from a cache filled with the same format . . . 37
6.1.6 Results from a cache filled with another format . . . 37
6.1.7 The speed and latency improvement of double caching over single caching 38
6.2 Future work . . . 39

7 Conclusion **41**

Bibliography **43**

Acronyms **45**

Appendices **47**

A Range requests done for the ISS format **47**

CHAPTER 1

Introduction

The internet started out as a means of sending messages from one place to another. Since then it has become much faster and is used by many different applications. It is still used to send messages, but because storage became cheaper and the internet became faster, it is now used for applications that require way more bandwidth. Because of it a revolution in the way we watch videos has taken place. Movie theaters and cable TV are not the only ways to watch movies and TV programs anymore. Streaming of videos over the internet has grown to an enormous industry that can be used to watch any kind of video imaginable. Lots of amateur videos are shared using YouTube, movies and TV shows can be watched easily using Netflix and even live TV is being moved to the internet, for instance by the BBC iPlayer.

This way of video streaming is not limited to normal computers. It is done on lots of different devices, like smartphones, tablets, set-top boxes, media centers or Smart TVs. Especially on mobile devices the network conditions can suddenly change drastically, for instance when changing the network connection from WiFi to a mobile network. With initial streaming technologies these changes caused buffering issues when the client was trying to get a high quality video using a low bandwidth network connection. This brought life to the notion of adaptive bitrate streaming (ABS), which is a way of streaming video that changes the quality of the video that is being watched, based on what the network connection can handle [1].

There are a couple of streaming solutions that implement ABS, most of those use HTTP as the underlying protocol [2]. One of the reasons for this is that HTTP is already used widely so it becomes possible to leverage the already existing infrastructure that is in place for other HTTP communication. A disadvantage of HTTP however is that it is a protocol for sending whole files and entire video files are typically quite large. This is why the solutions stream little sections of the video instead of the whole file. This means that the original video file should be cut in smaller files. They also need to store some extra data in those files, so a player will actually know how to play it. The solutions all have their own different way of adding this data.

When hosting video these different formats can cause quite a bit of trouble. The reason for this is that the different formats are not supported equally by the players used by different devices or browsers. Because of this, a good video host should have its videos available in a lot of different formats to make sure they can be viewed by every device and browser. The most obvious way of doing this of course, is generating the files in all the needed formats and then setting up a file server that simply serves all the little files. This works but takes up quite a bit of space since the same video is essentially stored multiple times.

Unified Streaming (USP) is a company that provides an easy to use solution for this problem. They do this by supplying modules for a set of popular HTTP servers, Apache, Nginx, IIS and Lighttpd. The way their module solves the problem of storing the video multiple times is by only storing the original video file and converting that on the fly to the requested format. This converting is also called transmuxing or trans-multiplexing, since it does not re-encode the video, but rather extracts the video stream and saves that in another format [3]. At the moment four different ABS formats are supported by the USP software. ISS Smooth Streaming (ISS), which is developed by Microsoft and is used in Silverlight based players [4, 5]. HTTP Dynamic Streaming

(HDS), which is developed by Adobe and is used in Flash based players [6]. HTTP Live Streaming (HLS), which is developed by Apple and used on the iPhone, iPad and Apple TVs [7]. The last format is Dynamic Adaptive Streaming over HTTP (DASH), which is developed as an ISO standard and it is, for instance, used in HTML5 video players [8, 9].

1.1 Traditional setups

The simplest way on the fly conversion can be used in a setup, is again a single server, that converts the original video file to the needed format for every request. However, in large streaming setups this is not enough for two reasons. The first one is that the storage server might be far away from the viewer, which means more time is spent waiting before watching a video. The second one is that the storage server can be easily overwhelmed with requests, because its just one server.

Using a content delivery network (CDN) with reverse proxy caching nodes is a simple solution to these problems, which can also used when hosting lots of content besides videos [10, 11]. Those nodes forward the request to the storage server and cache the response. If another request comes for the same content a node will just respond with the response in its cache. This works quite good, since a small percentage of the videos account for a large percentage of the views.

The big problem of this setup is that, just like with the very first setup, it stores the converted files instead of the raw files, only now in the cache instead of the storage server, which has two downsides in this case. Again, a server will essentially contain the same content multiple times, only in this case the cache server. The other downside, specific to this setup, is that the cache will also request basically the same content from the storage. Those requests mean more internal traffic and more waiting time for the viewer.

1.2 Approach

The problem described above is why this thesis will explore the possibility of using a "Late Transmuxing". This setup would do the conversion of the video directly on the caching nodes, instead of at the storage level. This means that the cache server would request segments of the original video file from the storage server, instead of the already converted files. This approach should solve both the extra storage problem and the internal traffic problem, because only the segments of original video file have to be stored and requested.

CHAPTER 2

Problem analysis

The introduction briefly described some of the server setups used for streaming video. This chapter will explain in detail what kind of setups are currently available and what their advantages and disadvantages are. The setups that will be looked into are the ones that convert the original video file on the fly to different ABS formats. The reason for this is that those setups already have a clear advantage over servers that simply store files in all formats, namely the space required to store the videos. To explain how each of the setups work, each step will be described, from the client requesting a segment of video in a specific format, to the client receiving that segment.

The different server setups consist of one or two servers. One server, which is present in every setup, stores the video files on disk, this will be called the storage server. The more advanced setups also contain a second server, which serves as a reverse proxy, this server will be called the proxy server. In the setups where a proxy server is used the client connects to the proxy server and the proxy server then requests content from the storage server that it needs to fulfill the request from the client. When caching takes place on the proxy server, it can also be called the cache server.

Lastly, on both of these physical servers will run an HTTP server to serve the HTTP requests. For illustration purposes a clear distinction between a physical server and the HTTP server that runs on that server needs to be made. This will be done by describing the setups as if Apache is used on the storage server and Nginx is used on the proxy server. Keep in mind however that this is just for illustration purposes, in practice any of the HTTP servers supported by USP can be used on any of the physical servers.

2.1 The single server setup

This setup was already briefly explained in the introduction and it is the most simple setup that will be covered. In this setup a client requests a segment directly from the storage server. Apache will then locate the video file that contains the requested segment. It will then read the parts of the file that are necessary to create the segment in the requested format. From these parts, it will then create the segment and send it back to the client. After the client receives it, it can be played by the player. For a schematic overview of this setup see figure 2.1 on the following page.

The main advantage of this setup is that is very simple to set up. It is just one server and all it has to do store files and run the conversion code. The main disadvantage is also that it is just one server. This means that it can easily be overwhelmed with requests when traffic is high, especially since it not only has to serve files, but also run the conversion code. Another issue that occurs because it is just a single server is that the time to reach the client can be quite long when the client is located far away from the server, for instance on another continent.

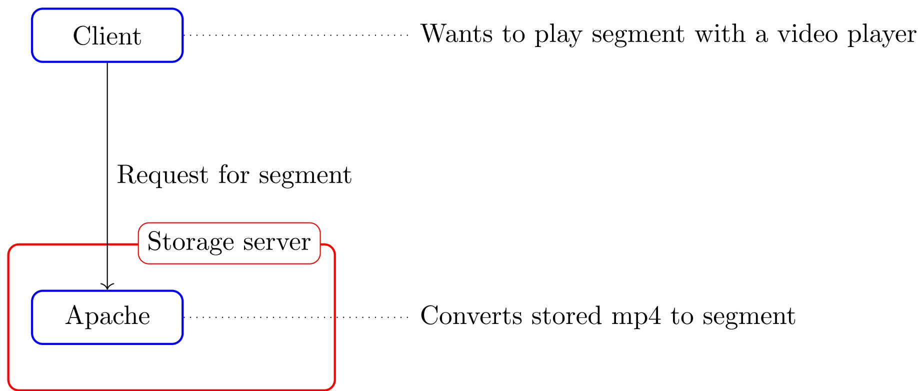

Figure 2.1: The single server setup

2.2 The CDN setup

Another setup that was already briefly introduced is the CDN setup. It is a quite a simple extension of the single server setup. In this setup the proxy server will receive the request for the segment from the client. It then passes that exact request along to the storage server. The storage server will then generate the segment from the original video file and send the segment back to the proxy server. The proxy server will then send the response back to the client, but it will also cache the response. That way, when a client requests the same segment in the same format the proxy server will be able to serve the segment directly from its cache, instead of requesting it from the storage server. For a schematic overview of this setup see figure 2.2.

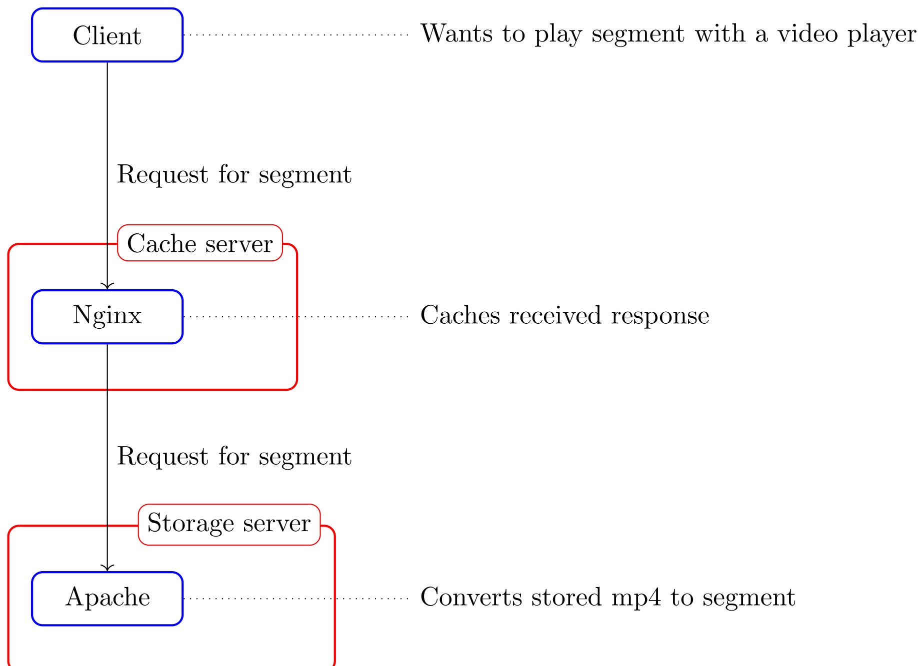

Figure 2.2: The CDN setup

Using this setup solves the issues present in the single server setup. The simple server setup was easy to overload with lots of requests. Caching should help quite a lot in this regard, since there normally is a set of videos that receives a lot of views and a set that receives (almost) none. Because of the caching, video files for popular videos only have to be converted once for each segment and format, instead of for every request. The latency issue is also mostly mitigated by the caching. By simply adding more proxy servers, spread around the globe, only the first

request for a segment will have high latency, the next request for that same segment only needs to travel to the proxy server.

One of the disadvantages of this setup is of course that it is less trivial to set up than the single server setup. It also has some other disadvantages. Because it caches the converted segments, it is essentially storing the same part of the video multiple times, only in another format, which is exactly the problem that on the fly conversion was there to solve. This time it is only doing this however for the most frequently watched videos, so it is less of an issue, but it is still something that is best kept to a minimum.

Another small disadvantage is caused by the proxy requests this server uses. A request that is not cached yet will actually take longer to return to the client than it would with the single server setup. This is because of the overhead of first traveling to the proxy server and then to the storage server. This is almost always a longer (and never a shorter) route then the route directly to the storage server. Because this longer route is taken only the first time a specific segment is requested and the next times the route will be shorter this is not such a big problem, but it still is something that should be noted.

2.3 The IsmProxyPass setup

Another setup that the USP software supports is the IsmProxyPass (IPP) setup. Just like the CDN setup, this setup extends the single server setup. However it does this in quite a different way. The key difference is that the USP software does not run on the storage server, but on the proxy server. This time, when the proxy server receives a request from a client, instead of passing it directly to the storage, it will send a series of HTTP range requests, which are generated by the USP software, to the storage server [12]. Range requests are a type of HTTP requests that do not request the whole file that the server would return for a normal HTTP request, it only requests a specific byte range of that file. These kind of requests are especially useful when only a small piece of a large file is needed.

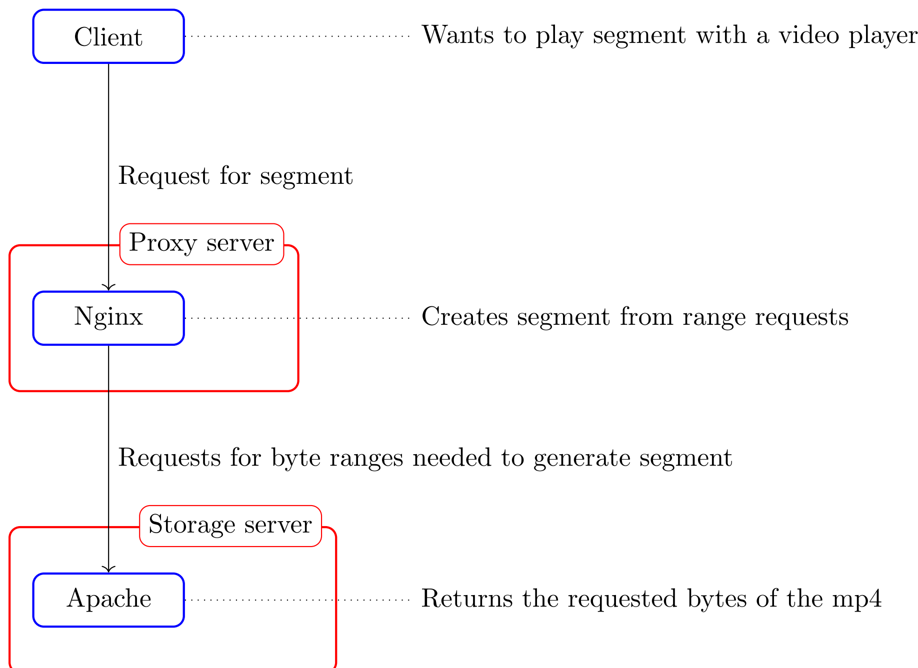

Figure 2.3: The IsmProxyPass setup

Upon receiving one of those range requests, the storage server will simply return the requested bytes from the video file. When the proxy server then receives those bytes it will use them to generate the segment requested by the client. After this is done the segment will be returned to the client. This all works by using USPs custom IsmProxyPass server directive and supply it a

URL at which it can reach the storage server [13]. For a schematic overview of this setup see figure 2.3 on the preceding page.

This setup shares some of the advantages of the CDN setup. It can easily be used to mitigate the storage overloading issue. By just adding more proxy servers the conversion work can be done by multiple servers. All that the storage server has to do is serve the byte ranges that the proxy servers need. The latency issue however, is still present. Since for every request of a client still needs to reach the location of the storage server, it does not matter that they are converted to range requests, they still need reach the storage eventually.

The issue with internal requests is again present and it is much worse than with the CDN setup. The first reason for this is that no caching takes place, so every clients request has to travel further, instead of just the first request for each segment. Another issue is that every request of a client spawns multiple range requests, which means even more overhead and thus more waiting.

Because of all this, it seems that this setup is only worse than the CDN setup. However, it does have one important advantage. Hosting large files on dedicated servers or VPSs can be quite expensive. This is why there are services that specialize just in hosting files, like Amazon S3. Services like this, which will be called dumb storage from now on, are quite popular for hosting large files, like videos, because it is relatively cheap. Dumb storage allows files to be uploaded and it will make them accessible over HTTP by supplying a URL. However, there is now way to access the server internals, like the HTTP server that is running on the server. This is no problem for static content, however, on the fly conversion makes the videos dynamic content. The USP software needs to be installed into the HTTP server itself, which means it is impossible to use on dumb storage. This setup solves this issue by separating the storage and the conversion tasks, which makes it possible to host the video files on dumb storage and use other servers for the conversion.

2.4 Combining the setups

Both the CDN setup and the IPP setup have their clear advantages. Both their disadvantages are clear as well. A combination between the CDN setup and the IPP setup that would have both of their advantages could be an ideal setup. However, currently no such setup exists. The clear advantage of a combination like this would be that it is possible to combine cheap dumb storage with the speed of CDN setup. However, another advantage is gained as well. In this combination the proxy server would be caching the byte ranges of the original video instead of the converted segments. This would mean that the same content is not stored multiple times in the cache, like in the CDN setup, since the same cached range requests can be used to generate a segment in a different format. This would also mean that the amount of internal traffic would go down since no requests would have to be done for the same segment in a different format. These advantages are why this thesis sets out to design such a setup and to test its performance against currently available setups.

CHAPTER 3

Design considerations

To build the proposed setup, some design decisions have be made. Most of those decisions concern the technologies to be used, but there are some other ones as well. This chapter will describe the options to choose from. It will also explain what the final decision for those choices is and why.

3.1 HTTP servers

The proposed setup consists of two different servers, the storage server and the proxy server. Both of these servers need to have an HTTP server to handle the incoming requests. Nginx and Apache are both very capable and popular HTTP servers. They are also both available for Linux and they are supported by the USP software. However, there are quite a different in some aspects and some of those aspects are important for the proposed setup.

3.1.1 General differences

There are a couple of general differences between Apache and Nginx that have already been thoroughly explored in a multitude of resources [14, 15, 16, 17]. These resources cover the differences very well and all that will be described here is a simplified summary of the differences. The core architecture of both servers is very different. Nginx uses an event loop to handle lots of requests in one process, while Apache mostly spawns more threads or processes to handle more requests. This means that with the same resources, Nginx can handle a lot more concurrent requests than Apache, since spawning threads and processes is quite expensive resource wise. Their take on dynamic content is also quite different. Apache uses modules that can be easily loaded into Apache by changing configuration files, these modules are, for instance, used to handle dynamic content. Nginx can also use modules, but they need to be added to Nginx at compile time, which makes it more of a hassle. That is why Nginx normally just receives the requests and sends them to other applications if content needs to be dynamically generated. In some setups Apache is even one of those other applications. These differences are why Apache shines as an application server, but Nginx is usually a good choice for a static file server or a reverse proxy.

3.1.2 USP integration

While USP supports both servers fully, because of the architecture of both servers it is easier to create a working setup with Apache. Apache can load modules for dynamic functionality easily, so this is used by the USP software [18]. Just download and install the module and enable it by adding some lines to the server configuration files. However, since Nginx is not designed to just load modules on the fly, the USP module for Nginx needs to be added at compile time [19]. This means it is not possible to just use the Nginx version supplied by the Linux distribution of

choice. This all makes it quite clear that the installation process for Apache is quite a bit easier than the one for Nginx.

3.1.3 Caching

Since the proposed setup depends heavily on caching in the reverse proxy, it is important that the server application does this well. Both Nginx and Apache have caching support [20, 21]. There is a key difference however, this is the better cache locking support in Nginx. Cache locking is a mechanism that can be used to reduce load on the storage server. Normally if a request comes in for some content and there is no cache for it, it will be proxied through to the storage server. This happens even if another request for the same content is already being proxied through but has not returned yet. If cache locking is enabled, the second request for some content will wait until the first request has returned and fills the cache. This can significantly reduce the load on the storage server, since the second request can just fetch the response from the cache. Nginx and Apache both support cache locking. However, Apache only uses it as a hint and it will still fetch from the storage if it receives a request with a header enable that tells it to do so. Nginx will always make a second request wait if cache locking is enabled, even with special headers.

3.1.4 Scripting

Both servers allow for low level scripting using Lua [22, 23, 24]. This can be very useful if complicated logic is needed to handle a request correctly. One big difference between the Apache version and the Nginx version is that the Nginx version can spawn subrequests, which can be very useful. For instance, one of the things subrequests allow, is that based on some checks in the Lua code the request can be handled by two entirely different Nginx paths. This way one can use the power of the complicated lua logic in combination with the simpleness of normal configurations.

3.1.5 The decision

Nginx is chosen as the HTTP server for the reverse proxy in the proposed setup. The better support for cache locking and scripting are the main reasons, since those could be useful for the actual setup. Its low memory footprint and speed are nice side effects, but they are not essential for the setup itself. Which type of HTTP server is used by the storage server is not really important for this specific setup, since they can both serve static files and support range requests out of the box. Apache will be chosen for the setup that will be tested, not because it is better for this setup, but because it is easier to enable the USP software for it. This will make it easier to test against other setups that need on the fly conversion on the storage server, like the CDN setup.

3.2 What responses to cache

The proposed setup is basically a version the IPP setup with caching enabled. Because of this, it is important to decide what it is that needs to be cached exactly. There are two types of responses that could be cached on the proxy server, the range request responses it receives from the storage server and the converted segments it sends to the client. The range requests responses are most important to cache. These are most important since they cause network traffic between the proxy server and the storage server, which introduces latency.

The responses the proxy server sends to the client might also be useful to cache. In the CDN setup these responses were mostly cached because of network latency. However, when already caching the range requests in the IPP setup, network latency should not be an issue, since the responses can be generated on the fly from the cached range requests. Caching the result of this conversion should reduce the CPU load on the proxy server, since the segment does does not need to be generated again. Caching the converted segments still has the storage disadvantage it had with the CDN setup. The same segments are essentially cached multiple times and this

is even worse when also caching the range requests, since that means segments are cached one extra time in comparison to the CDN setup.

3.2.1 The decision

Currently it is unclear what the effect of caching each type of response has on the performance of the setup. This is why the final implementation will come in two flavors. A setup where all of the previously described responses are cached and one where only the range requests are cached. A third flavor would be one where only the responses to the client are cached, but since this flavor would basically be the same as the CDN setup it is not necessary when also testing against that setup.

3.3 Caching range requests

Since the IPP setup uses range requests it is also important that those can be cached. This is not directly supported in Nginx or Apache. The reason for this is that it is not exactly trivial to do. Most caches work by supplying it with some data to cache, together with a key by which the data can be retrieved later on.

Ideally the content would be cached using the key of the full file and some info would be saved about what range is actually cached. Then when another request for a range comes in it would check in the cache if some of the requested bytes are in there. If not all of them are there it should request just the bytes that are not cached from the upstream server and then add those to the same cache entry for future use. A cache that works like this hasn't been implemented by Apache or Nginx and quite possible by no HTTP server in existence. However, our proposed setup is not the first that would benefit from caching range requests. So some suboptimal solutions have been proposed for different use cases.

3.3.1 Ignore range

The most simple one, which is employed by Nginx by default, is removing the range request header from the request when sending it upstream. This will request the whole file instead of just the requesting byte range. Nginx will then cache this and return the whole file every time a range is requested. According to the RFC this behaviour is allowed, but this it is clearly not desired behaviour, since the whole point of range requests is that requesting the full file is not necessary.

3.3.2 Match exact range

A simple solution to this, proposed on the Nginx forums, is to force Nginx to request the range from upstream and then adding the requested range to the cache key [25]. This basically means that a range request is handled by the cache as if it were a separate URL. The main downside of this solution is that when overlapping byte ranges are requested they will both be cached in full. This means this solution works fine when the ranges that get requested are always the same. However if the requested ranges are different almost every time, this solution is not acceptable since a lot of the cache will contain the same bytes, because of the overlap in the byte ranges.

3.3.3 Ranger

Ranger extends further on the previous solution and tries to solve the overlap problem [26]. Every time a range request comes in, instead of sending it directly to the upstream server, it generates other range requests that together contain all the bytes requested. These new range requests always have the same block size and they also start at the same byte position. These requests will then be sent to upstream and after a response comes back they will be cached instead of the original request. This way, even though the initial range request can be any range and can overlap with others, the requests sent to upstream will always match previously sent requests, which makes sure that cache hits occur every time part of a range is requested that

was requested before, even though it's not exactly the same. There are only two small issues with this approach. The first one is that more data is requested from the upstream server than is actually needed to fulfill the request. However, this is probably not too severe when the block size is chosen small enough. The second issue is that when large ranges are requested, a lot of requests need to be send to the upstream server, because the large request gets split into a lot of smaller ones. Which could mean some unnecessary overhead.

3.3.4 The decision

At this point it is clear that the first approach for caching range requests is not sufficient, since that would mean the whole video would have to be sent to the proxy server to send a small segment of that video to the client. The second approach is one that shows great promise for the proposed setup. If the timing and length of segments sent for each format are the same, it would probably mean that the ranges requested are exactly the same as well. This would mean that the caching with this approach would just work. However, if after preliminary testing it is shown that a lot of the ranges are (slightly) different, using ranger might be preferable over the second approach.

CHAPTER 4

Implementation

The newly developed setup will be called the late transmuxing (LT) setup, because it transmuxes the video in the storage instead of in the cache. The implementation of the LT setup consists of configuration files for the HTTP servers running on the proxy server and the storage server. What it implements, stating it very simply, is a version of the IPP setup with caching enabled. This chapter will explain at a high level how the implementation works. The implementation described is the flavor that caches all types of responses, not just the range requests. The reason for this is that the other flavor can simply be achieved by disabling the caching of the responses to the client.

4.1 Initial implementation

With everything from Design considerations in mind the initial LT setup was implemented in the way described here. The setup should consist of two servers, a proxy server that runs the USP code and a storage server that serves byte ranges. The proxy server should be able to cache the responses it sends to the client and the responses it receives from the storage server. The USP code does not support caching natively. This can easily be solved by letting Nginx make requests to itself. It can then use its normal proxy and caching mechanisms to pass requests on to the actual destination and cache the responses it gets before returning those to the original requester.

A schematic overview of the setup described next can be found in figure 4.1 on the next page. First, client requests a segment from the proxy server. Nginx receives this request and passes it through its caching proxy to the IsmProxyPass handler. This handler is setup to request the byte ranges from a second proxy running in Nginx. It then uses the USP software to generate those range requests and requests them from that proxy. The second proxy then passes those requests on to the storage server. The storage server will then receive the requests and return the requested byte ranges.

When the second proxy in Nginx receives a response to one of its range requests it caches that response. This is done by using the second method described in section 3.3. Apart from caching the response the proxy will send it back to the IsmProxyPass handler. Apache will then generate the originally requested segment and return that back to Nginx. Nginx will then cache the segment and return it to the client.

The outcome of the initial tests of this setup were as expected. As it turns out however, in its current form the setup has one big problem. When testing this setup with more concurrent connections than the amount of processes that Nginx is configured to use it results in deadlock. The reason for this is that the IsmProxyPass directive supplied by USP spawns the range requests in a blocking way, instead of in the non blocking way that Nginx normally spawns requests. This causes the threads that spawn them to wait until they get a response from the upstream server. However, since the upstream server is the same server, it will only be able to do so when not all the threads are blocked. When using enough concurrent client requests all available threads will

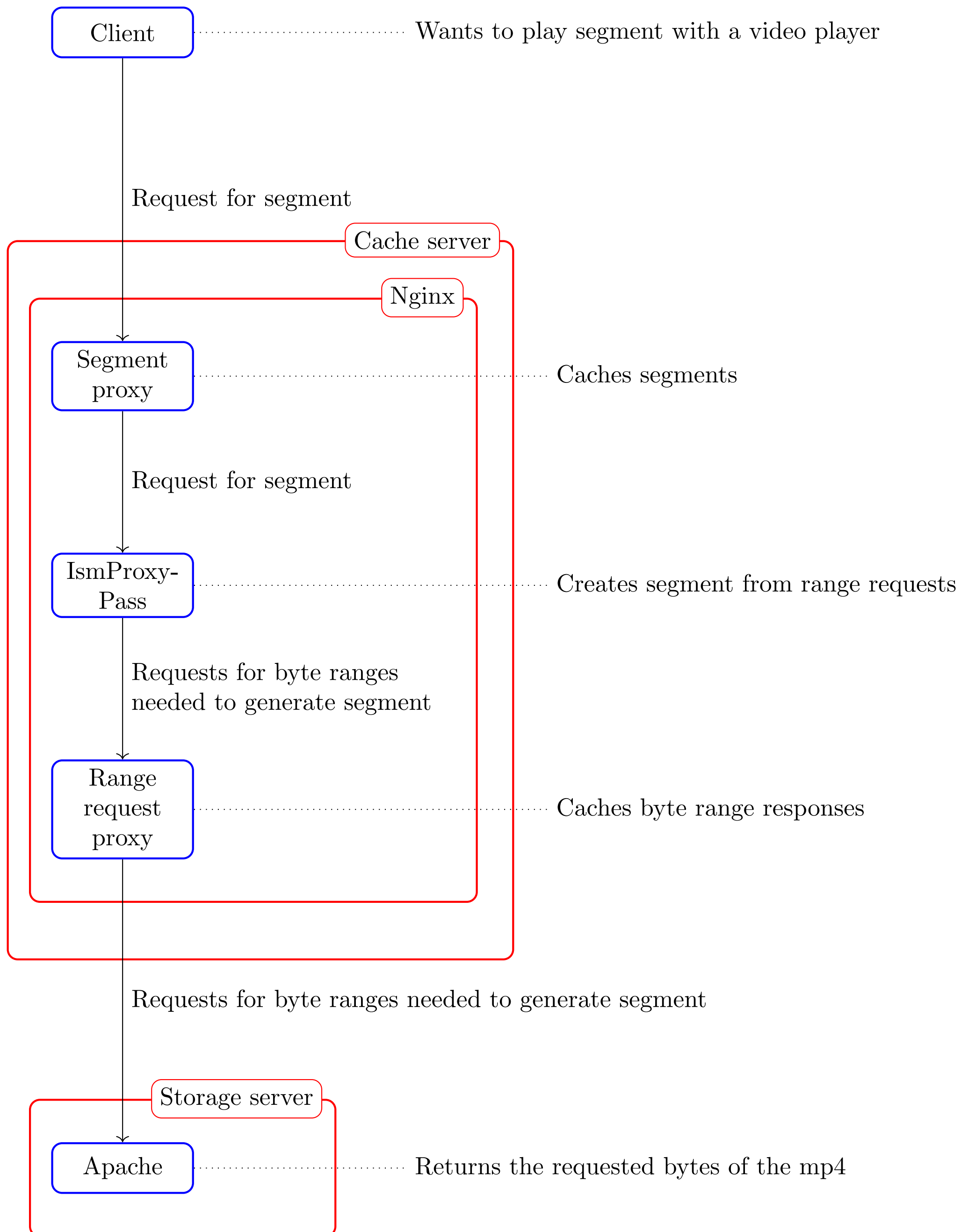

Figure 4.1: The initial late transmuxing setup

get into the waiting state, so no more threads are available to serve the requests that would get them out of this state.

4.2 Final implementation

There are a couple of possible solutions to the deadlock problem, but by far the easiest is moving the blocking code outside Nginx. In the final implementation this is done by not only running Nginx on the proxy server, but also Apache. The only task of Apache is to run the IsmProxyPass code. This way all that Nginx does is proxying and caching. Once a request of a client comes in, it will pass through Nginx to Apache on the same server. Apache will then generate the range requests and those will pass through Nginx again, so it can cache their responses. Nginx will then send them to the storage server and then the response from the storage will follow the chain back. And just like in the initial implementation when a response passes a proxy in Nginx it will be cached. For a schematic overview of this setup see figure 4.2 on the following page.

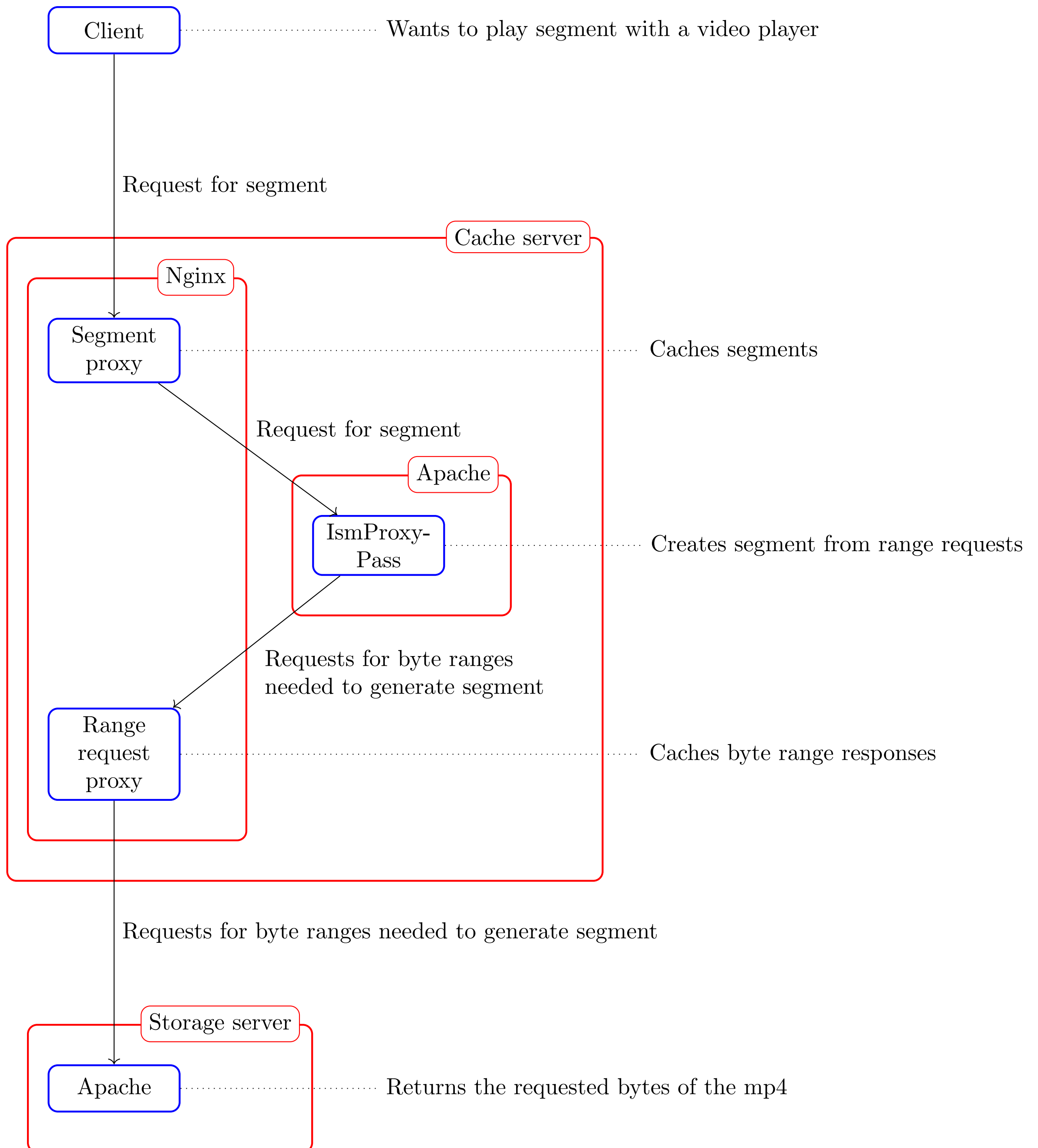

Figure 4.2: The final late transmuxing setup

CHAPTER 5

Experiments

This chapter will explain the experiments that have been done to compare the LT setup to other available setups. It will also show the results of these experiments.

5.1 Setup

5.1.1 Virtual machines

To create the proposed setup a client, a proxy server and a storage server are needed. This could be have been done by using actual physical servers for every type of machine. However, to make the development process easier the different machines where simulated using virtual machines. VirtualBox was used to create the VMs. One VM was used as the proxy server and a second one was used as the storage server. The host operating system was used as the client. Since the machines have to communicate with each other, a separate network interface was set up to allow communication. Both VMs have been set up with the same virtual hardware, they are both allowed to use 2 GB of RAM and one CPU core.

Comcast

Since the VMs will be running on the same machine there are almost no network limitations. That is why artificial network throttling is needed to simulate more realistic real world environments. Comcast is a command line tool that provides a simple interface over standard Linux tools to throttle the network [27]. Using Comcast the bandwidth between the proxy server and the storage server is throttled down to allow a maximum of 100 Mbit/s.

5.1.2 The different setups

The VMs contain six differently configured setups that will be tested by a variety of tests. These setups can be split into old setups, new setups and control setups. The old setups consist of the CDN and IPP setup, described in the chapter Problem analysis. The new setups consist of the two flavors, of the newly implemented late transmuxing setup. The flavor that only caches the responses to the range requests will be called LT-single and the flavor that also caches the responses to the client will be called LT-double. The control setups are setups that are purely there to test the effect of the actual caching in the LT and the CDN setups. These setups have all caching disabled, but the rest is the same and they will respectively be called LT-nocache and CDN-nocache. A simple overview of the setups can be found in table 5.1.

5.1.3 The tests

The performance of each setup is measured when downloading all the segments of a whole video, in the highest quality available and in all the formats supported by the USP software. All the

segments of the video in a specific format will be referred to from now on as "the video".

Generating requests

It is also interesting to measure how the different setups perform under different loads. Because only one client is used, it is necessary that that client is able to generate a significant amount of requests in a short period of time. Otherwise the differences in performance under heavy load cannot be measured. Wrk is a HTTP benchmarking tool that will be used to do this [28]. It can be supplied with an URL, a duration and the amount of concurrent connections. It will then send requests to that URL, for the provided duration and with the specified amount of concurrent connections. Afterwards it will output statistics about the network performance. The different tests are, in this case, run with 1, 2, 5, 10, 25 and 50 concurrent connections.

Wrk also provides a Lua scripting interface, which allows for more complicated logic to be used when sending and receiving requests. This interface is used by the testing setup to fire requests for all the segments of the video. This is done by looping over the contents of a file that contains all the URLs for the different segments of the video in a specific format. The files for the different formats have been generated by letting a player play the video from start to end and saving the requests. The player used for this was the Unified Capture tool created by USP, which impersonates a player to save an online video to a file [29].

Different requests for different formats

Because the different ABS formats differ quite a bit in some aspects, they also need different amounts, of different types of files, to play the whole video. There are four different types of files, used by the formats. There are playlist files, these tell the player what further files to request to actually play the video. Furthermore there are video files, audio files and combined audio/video files. Both DASH and ISS use separate video and audio files. For our specific video, both request 184 video files and 147 audio files. HDS and HLS use files that combine the video and audio into one file, for each segment, both formats use 184 of these files. Some differences also exist in the amount of playlist files, ISS and HDS only need one to play the video, DASH uses three and HLS uses five. This information is not necessarily useful to set up the tests themselves, but it might be useful when interpreting the results.

Measuring performance

Performance of a setup will be measured by a six kinds of characteristics. The first three characteristics are simply measured by wrk. The first of these is the amount of requests that are handled per second. The second is the amount of Megabytes received per second and the third is the average time it takes for a request to return a response. The fourth characteristic is the storage space that the cache directory takes up on the proxy server. The last two characteristics involve the traffic between the proxy server and the storage server. That traffic is measured in

Name	Type	What is cached	Which server converts the original video file	Figure
CDN	Old	Converted segments	The storage server	2.2
CDN-nocache	Control	Nothing	The storage server	2.2
IPP	Old	Nothing	The proxy server	2.3
LT-single	New	Byte ranges of the original video	The proxy server	4.2
LT-double	New	Byte ranges of the original video and converted segments that are sent to the client	The proxy server	4.2
LT-nocache	Control	Nothing	The proxy server	4.2

Table 5.1: An overview of the characteristics of the different setups

the amount of requests and the amount of bytes sent. These are simply measured by parsing the log files on the storage server.

Cache conditions

Performance tests will be executed on the different setups with three different cache conditions. The first one is where the cache is empty, also called cold. The second one is where the cache is already warm, from downloading the video in the same format that is getting requested. The last one is where the cache is also warm from downloading the same video, but in another format than the one that is requested.

The tests for the first and the last situation simply try to download the video once, as fast as possible. The tests for the second situation try to download the video as often as possible in a 30 second time frame. The reason this difference exists, is that the first and last situation both change to the second situation when the video has been downloaded once, since the cache has become warm from downloading the video in that format.

Since the different cache conditions should have no impact on the setups without caching, the IPP, the CDN-nocache and the LT-nocache setup have only been executed for the first situation. This is done to reduce the time that is needed for a test run, so more useful tests can be executed in the same time.

5.2 Results

The results of the different measurements are shown separately for the three different cache conditions. The transfer speed and latency measurements are plotted against the amount of concurrent connections. Important to note is that the concurrent connections will be plotted on a logarithmic scale, since the values used increase almost exponentially and this scale will make sure the data points are spread more equally. For the cache usage and internal traffic measurements this is not the case, since the amount of connections have no effect on these statistics. In all plots the confidence intervals for each data point is plotted as well, if these cannot be seen it means that they are very small. Not only the plots are shown, but the important information that can be gathered from the plots is noted as well.

5.2.1 Cold cache

First, the data of the cold cache tests will be shown.

Transfer speed measurements

In the figures 5.1 and 5.2, which show the transfer speed measurements, a couple of things should be noticed:

1. The CDN type setups perform best in almost all of the transfer speed measurements.
2. All setups reach a maximum transfer speed when the amount of concurrent connections is increased.
3. The CDN type setups already reach a maximum in transfer speed after two connections.
4. Unlike the other setups, the IPP setup reacts very irregularly to an increasing amount of connections.
5. Some setups behave quite similarly to others. The CDN type setups perform roughly the same; the caching LT setups do this as well. The IPP and the LT-nocache setup reach the same maximum and they also perform quite the same when using one connection. However, with two or five connections they perform quite differently.
6. The maximum reached by the caching LT setups is almost as the same as the maximum from the CDN type setups, although it is reached only with more concurrent connections.

7. Most setups perform similarly for the different formats when comparing MB/s, but when comparing requests per second the performance for HDS and HLS is worse that the performance for DASH and ISS.

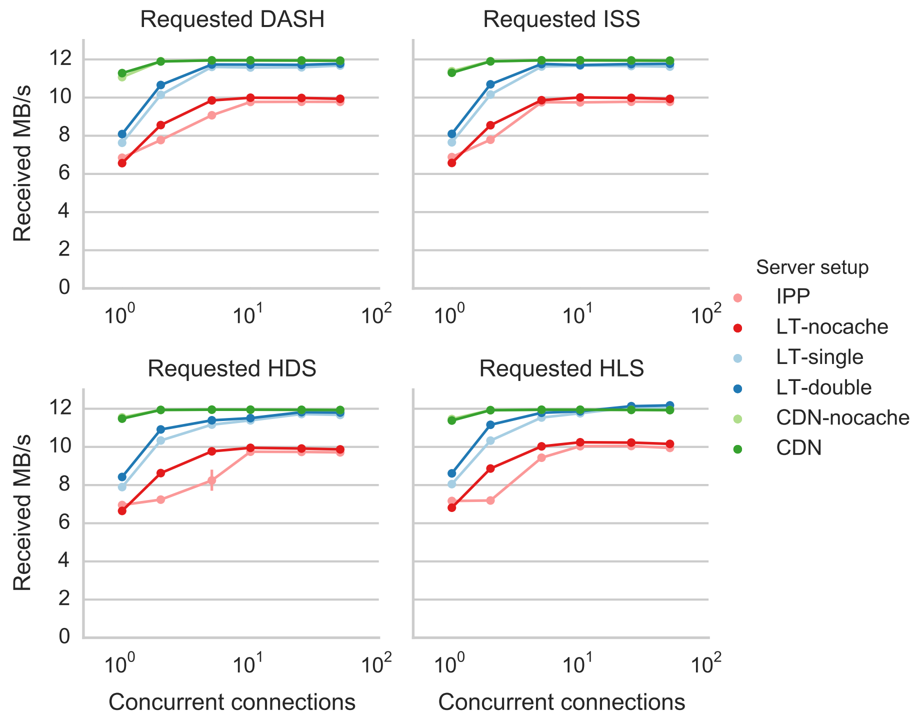

Figure 5.1: Received MB/s with a cold cache

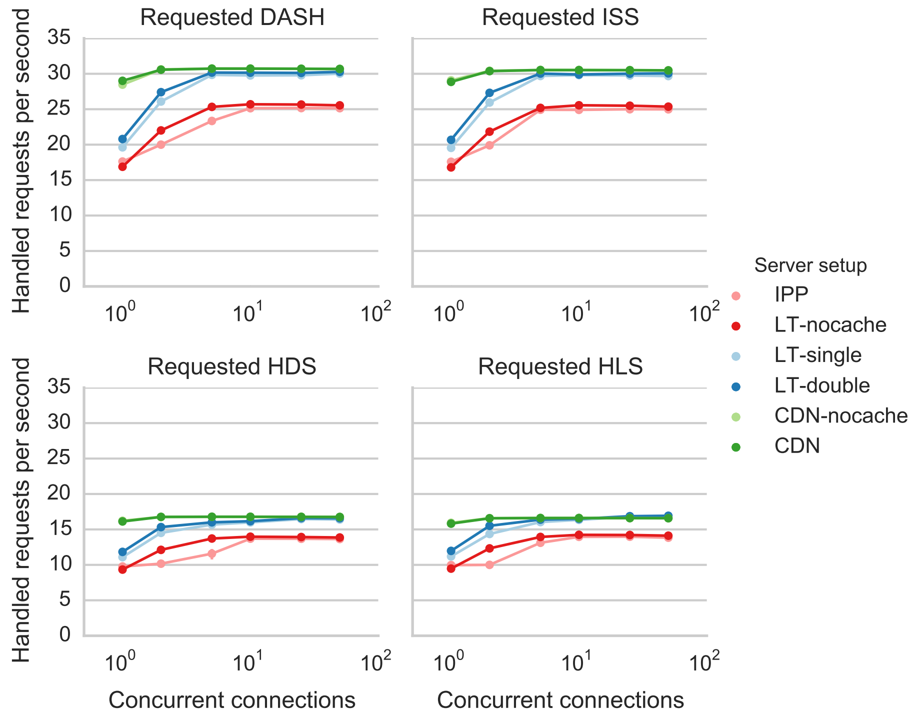

Figure 5.2: Handled requests per second with a cold cache

Latency

Figure 5.3 shows the difference in latency and how the latency is influenced by increasing the connections. Important to note is that in this case plot the Y axis also uses a logarithmic scale, this has two reasons. First of all, this way the relative difference can be seen better when looking at the lower connections. The second reason is that linear trends can be spotted better this way.

1. The worst and best performing setups are again respectively, the IPP setup and the CDN type setups.
2. With one connection the CDN setups outperform all the other setups.
3. With enough concurrent connections the caching LT setups again perform about as good as the CDN type setups.
4. The latency seems to increases linearly with the amount of connections.

Cache usage

The cache usage measurements in figure 5.4 show a only a couple of important things:

1. The setups that use no caching still use a small amount of storage for cache.
2. The LT-single setup and the CDN setup use about the same amount of cache.
3. The LT-double setup uses almost double as much cache as the LT-single and CDN setup.

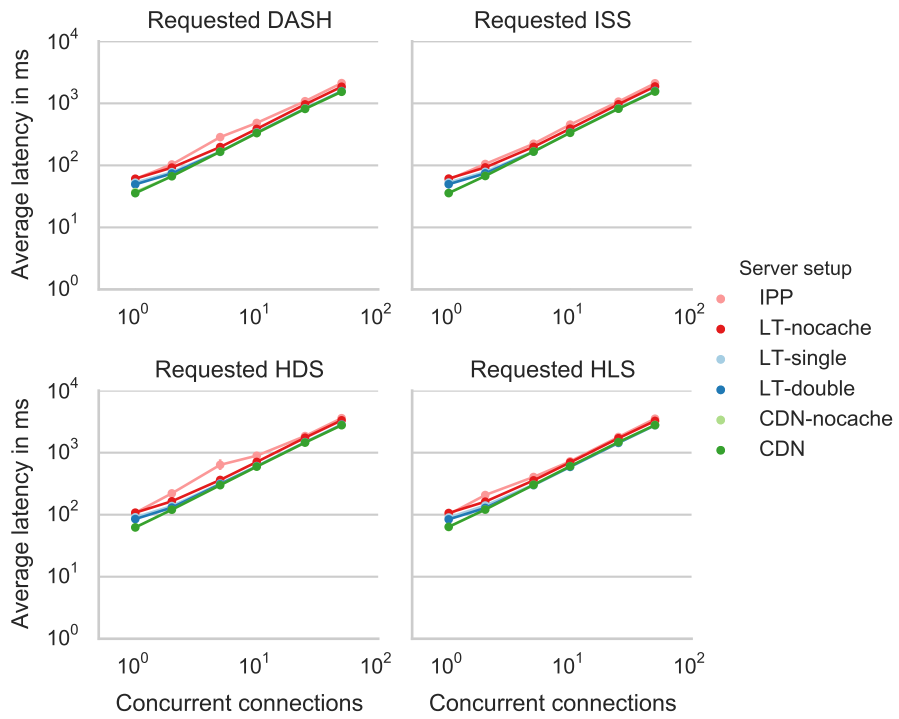

Figure 5.3: Average latency in ms with a cold cache

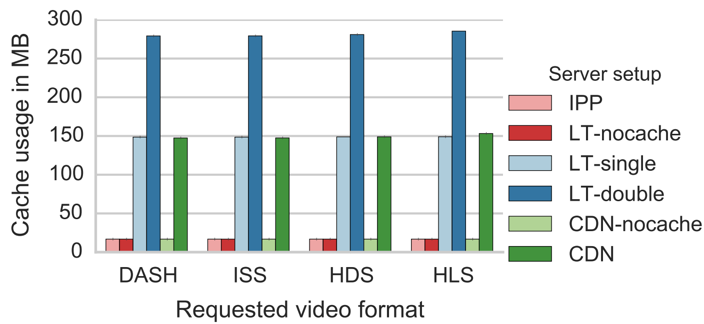

Figure 5.4: Cache usage in MB with a cold cache

Internal traffic

The internal traffic statistics in figure 5.5 and 5.6 show these things:

1. The IPP and LT-nocache setup perform the same; the CDN type setups do this too, just as the caching LT setups.
2. In both statistics, the IPP and LT-nocache setup perform worst.
3. The CDN type setups and the caching LT setups perform about the same when comparing the amount of MB that is sent internally.
4. The IPP and LT-nocache setups send more than twice as many internal requests as the caching LT setups send.
5. The caching LT setups send quite a bit more internal requests than the CDN type setups. When compared to the CDN type setups, the caching LT setups send about double the amount of requests for DASH and ISS and about three times the amount of requests for HDS and HLS.
6. The amount of requests for HDS and HLS is higher than the amount of requests for DASH and ISS is higher when looking at the IPP and LT type setups, but it is lower when looking at the CDN type setups.

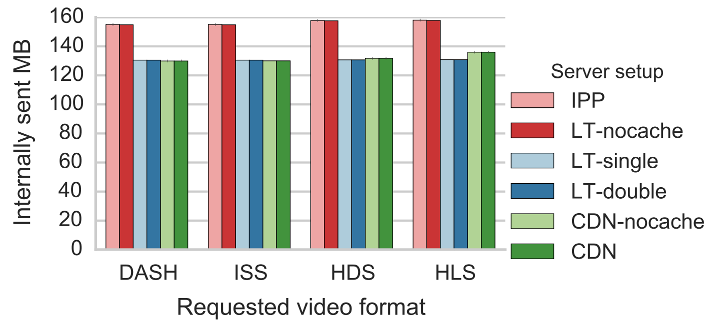

Figure 5.5: Internally sent MB with a cold cache

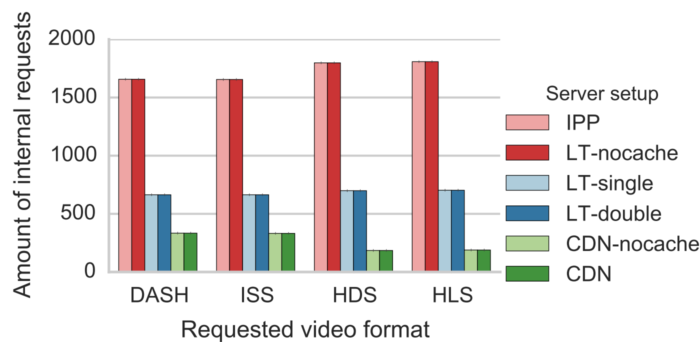

Figure 5.6: Amount of internal requests with a cold cache

5.2.2 Cache filled with the same format

Apart from explaining what the plots in this section show, this section will also compare the plots to the equivalent cold cache plots.

Transfer speed measurements

The figures 5.7 and 5.8, that show the transfer speed statistics, show this information:

1. The LT-double and the CDN setup perform roughly the same in all cases.
2. DASH and ISS perform roughly the same for all setups and HDS and HLS perform mostly the same as well.
3. When increasing the amount of connections the LT-double and CDN setup first perform better, but at some point the performance will drop drastically again.
4. The LT-single setup performs significantly worse, especially with a low amount of concurrent connections.
5. All setups perform better with a filled cache than they did with a cold cache, although performance gains achieved by the LT-single setup are much smaller than those that are achieved by the LT-double and CDN setups.

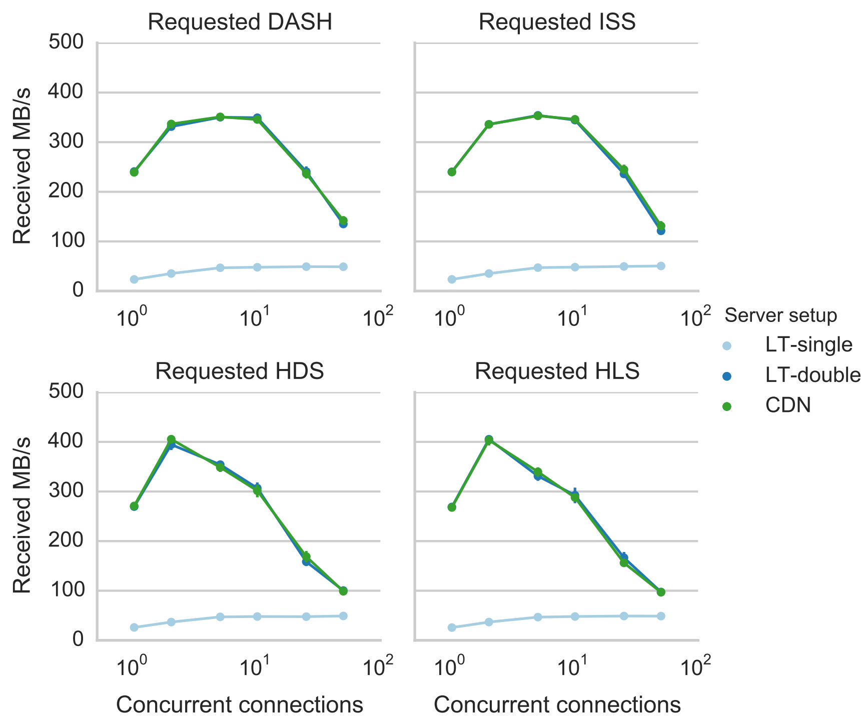

Figure 5.7: Received MB/s with the cache filled with the same format

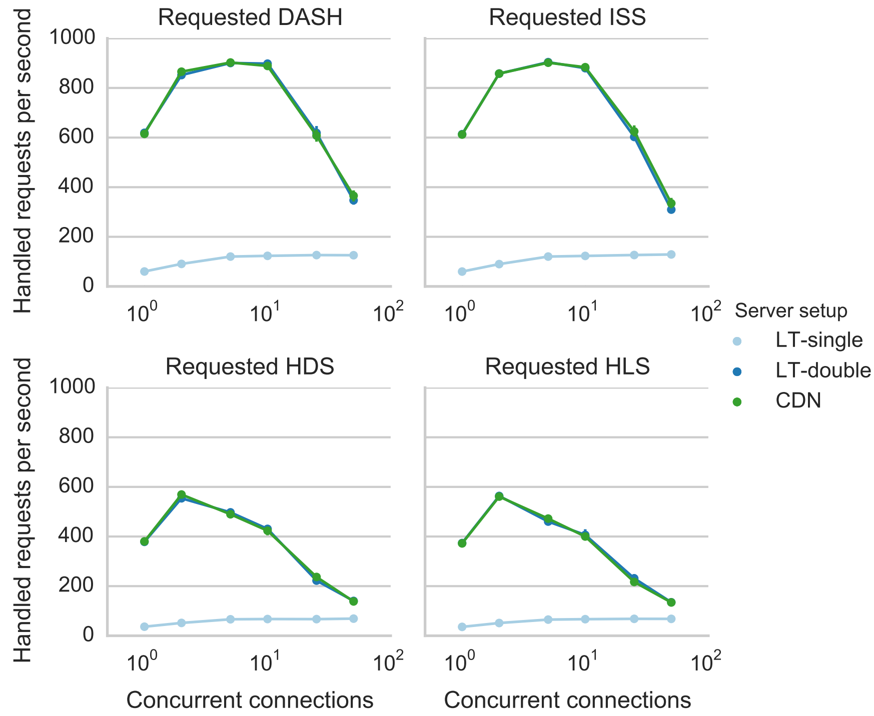

Figure 5.8: Handled requests per second with the cache filled with the same format

Latency

All the observations that were made for the transfer speeds also hold true for latency, which can be seen in figure 5.9. The only difference is that with two connections the confidence intervals for CDN and LT-double are exceptionally large for some of the formats.

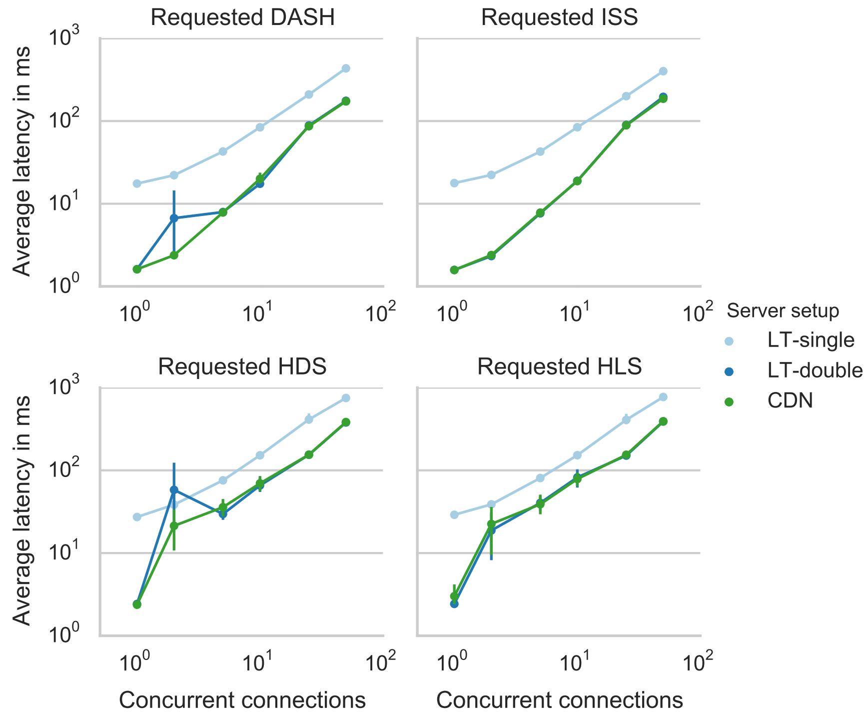

Figure 5.9: Average latency in ms with the cache filled with the same format

Cache usage

The only thing to gather from figure 5.10 is that the cache usage is exactly the same as the cache usage with a cold cache.

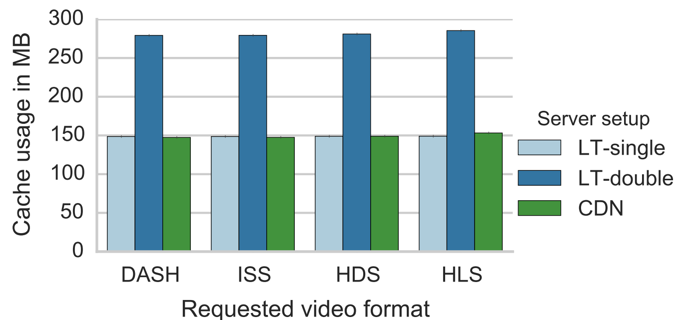

Figure 5.10: Cache usage in MB with the cache filled with the same format

Internal traffic

The figures 5.11 and 5.12 show quite clearly that no internal requests are sent for any of the setups.

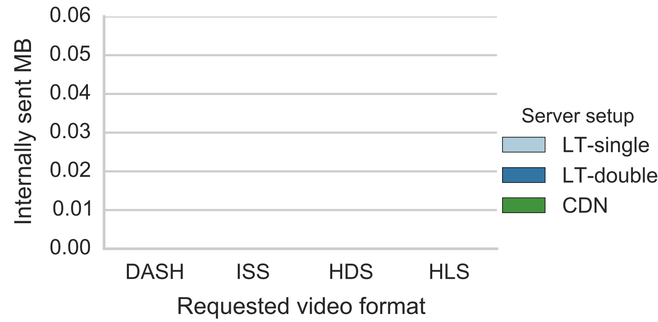

Figure 5.11: Internally sent MB with the cache filled with the same format

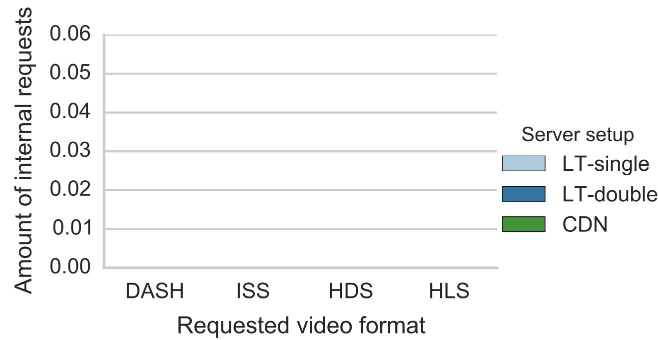

Figure 5.12: Amount of internal requests with the cache filled with the same format

5.2.3 Cache filled with another format

This section will show the performance of the different setups when running them after the cache has been filled with the video in another format. It is important to note that the following plots are a bit different from the previous ones. These plots compare the performance of the setups for the different video formats like before, but they also compare it to the different formats that were loaded in the cache. To plot this in a nice and easy way there are also plots that supposedly show a format being requested after itself. These plots are left empty however, since this would mean the cache was filled with the same format instead of another format. Lastly, to make it easier to specify what format was in the cache and what format was requested, this will further be called the requesting order. When a line says that DASH was requested after ISS, it means that ISS was already in the cache and DASH was requested.

Transfer speed measurements

In the figures 5.13 and 5.14, the diagonals are left empty, since they would show the data shown in the previous subsection. In the other subplots some interesting things can be noticed:

1. The amount of requests per second are a lot lower for HDS and HLS. This was also the case when requesting videos with a cold cache.
2. In all of the cases, the LT setups perform better than they did with a cold cache. The CDN setup performs roughly the same however.
3. Both the LT setups perform better than the CDN setup in all of the cases.
4. When comparing received MB/s for the LT setups it seems that the requesting order of two formats does not matter much. Complementary plots are very similar, for instance requesting DASH after ISS performs about as well as requesting ISS after DASH

5. When again comparing received MB/s for the LT setups, it shows that requesting HLS after HDS, DASH after ISS and both of these the other way around, all perform roughly the same. And any of the other requesting orders perform roughly the same as well. The first set of these perform best for both the LT setups.

6. When using five concurrent connections all the setups reach a maximum in both performance measures.

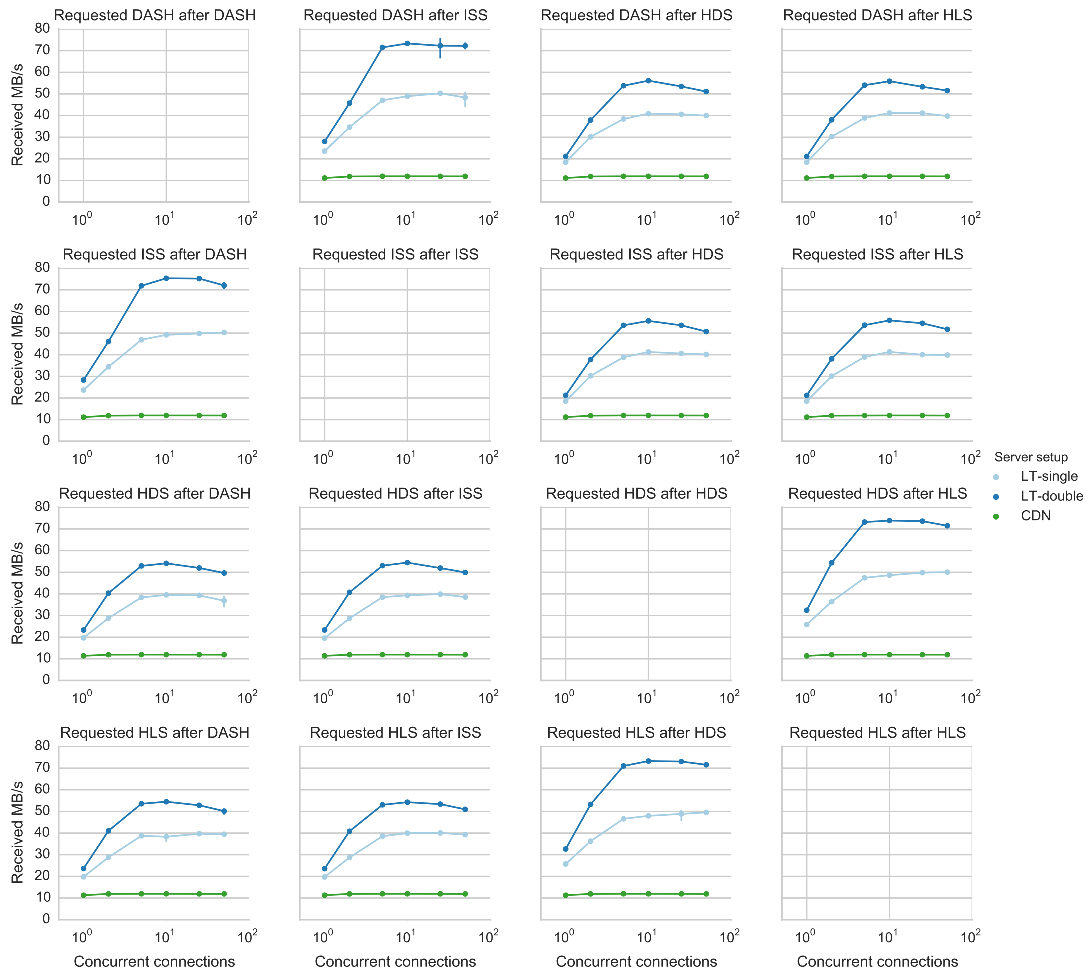

Figure 5.13: Received MB/s with the cache filled with another format

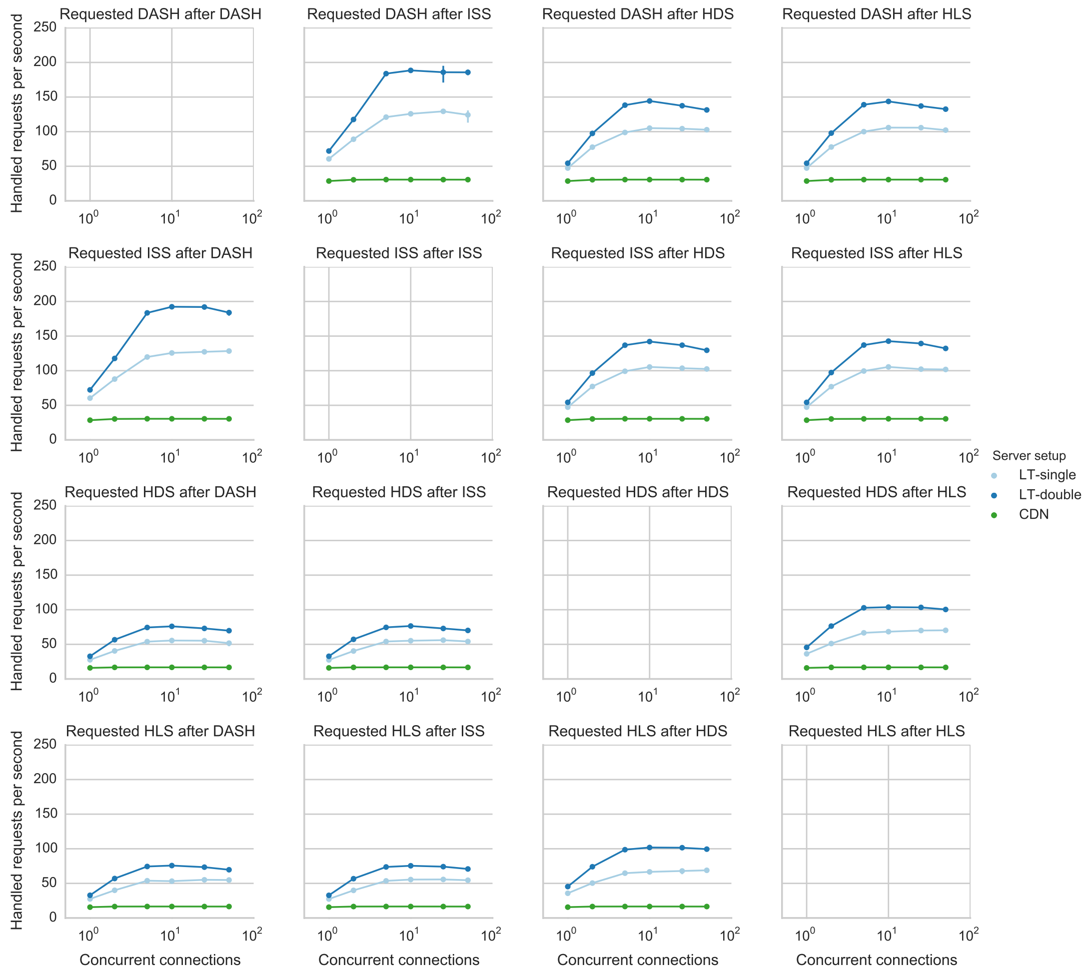

Figure 5.14: Handled requests per second with the cache filled with another format

Latency

The latency plot in figure 5.15 mostly shows the same characteristics as the requests per second plot in figure 5.14:

1. Again the LT-double setup performs better than the LT-single setup, which then outperforms the CDN setup again.
2. The latency for the CDN setup is roughly the same as it was when requesting the same format with a cold cache.
3. The LT setups perform better than they did when the cache was cold.
4. HDS and HLS perform worse than ISS and DASH.
5. The tests where DASH was requested after ISS, or ISS after DASH, perform best.
6. The tests where HLS or HDS were requested after DASH or ISS perform worst.

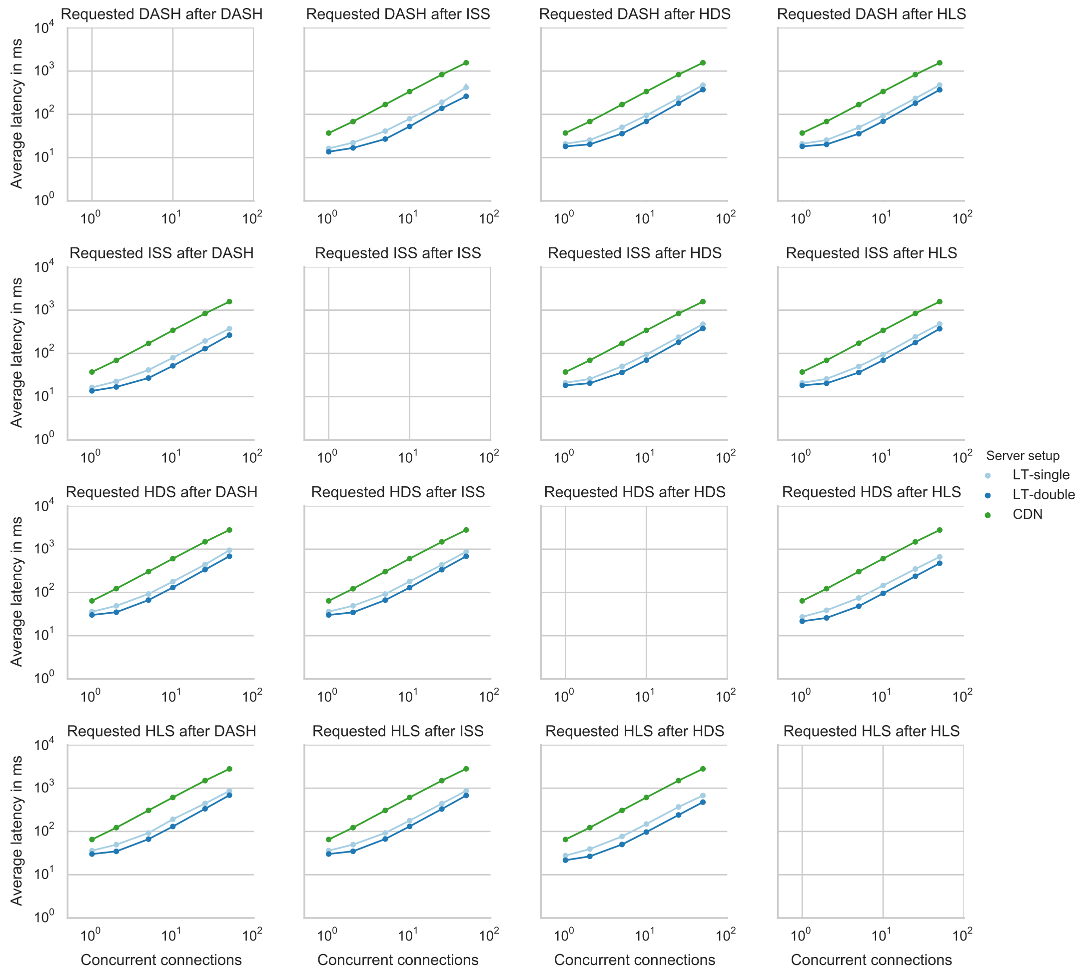

Figure 5.15: Average latency in ms with the cache filled with another format

Cache usage

First, to reiterate, the bars that represent requesting content after itself are empty not because they are zero, but because they would show the results shown in the previous subsection. The other bars show this however:

1. The cache usage for the LT-single setup is roughly the same as it was with a cold cache. For the LT-double setup the cache usage increased by about half and for the CDN setup it almost doubled.
2. The LT-single setup uses the least amount of cache in all cases.
3. The CDN and LT-double setups use respectively, about two and three times the amount of cache that the LT single setup uses.
4. In some of the cases the LT-single setup uses a bit more cache than it did when requesting the same format with a cold cache. These cases are HDS and HLS after DASH and ISS and the other way around. When requesting DASH after ISS or HLS after HDS or both the other way around the cache usage stays the same.

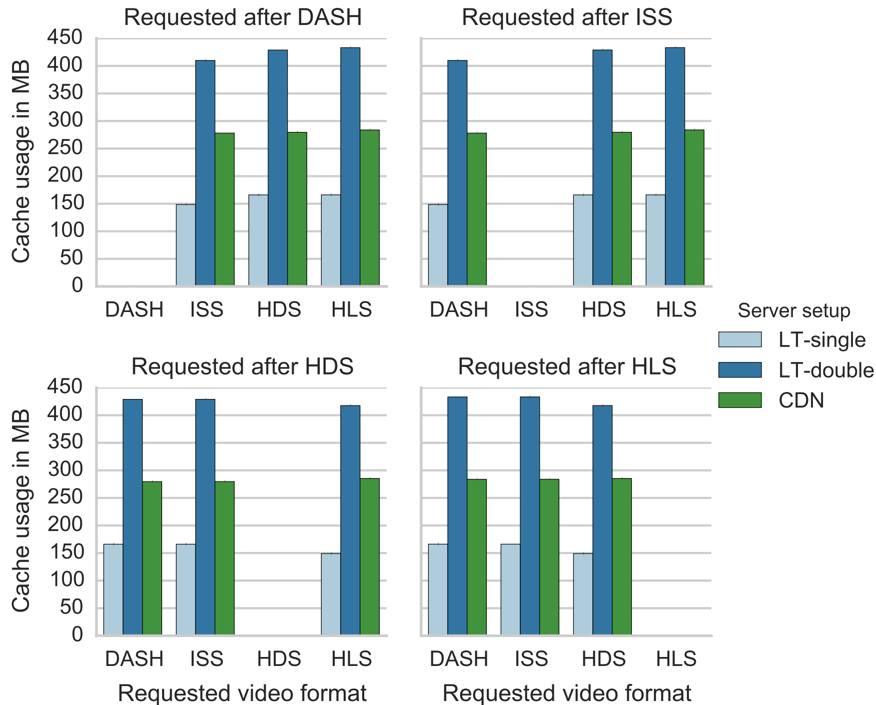

Figure 5.16: Cache usage in MB with the cache filled with another format

Internal traffic

The information that can be gathered from the internal traffic statistics is this:

1. The amount of internal traffic generated by the CDN setup is about the same as it was for the same format with a cold cache.
2. There is no difference in the amount of internal traffic between the two LT setups.
3. Looking only at the LT setups, when requesting DASH after ISS, ISS after DASH or HDS after HLS no internal data is sent.
4. Looking only at the LT setups, when requesting HLS after HDS only a very small amount of requests is sent internally.
5. Looking only at the LT setups again, when requesting HDS or HLS after DASH or ISS about 150 internal requests are sent. When DASH or ISS are requested after HDS or HLS about 180 internal requests are sent. The amount of data that is sent in all these cases is all roughly the same.
6. Comparing the LT setups with the CDN setup in the cases mentioned in the previous item. When HDS or HLS are requested, the amount of requests is about the same for both types of setups and when requesting DASH or ISS, the amount of requests for the LT setups is about half of that of the CDN setup. When comparing the amount of data sent internally, the LT setups send about six times less bytes than the amount of bytes that the CDN setup sends.

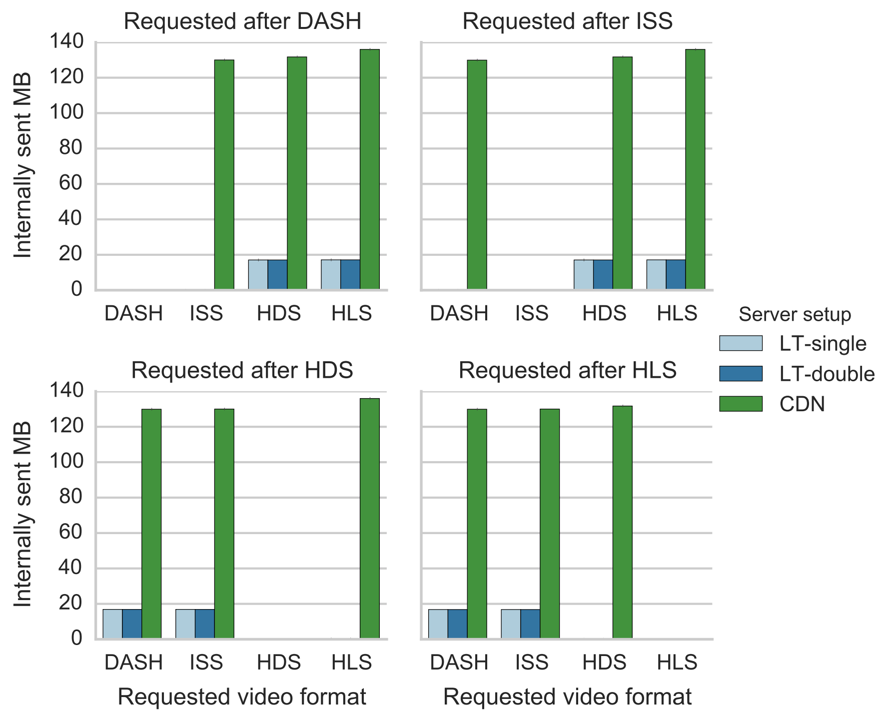

Figure 5.17: Internally sent MB with the cache filled with another format

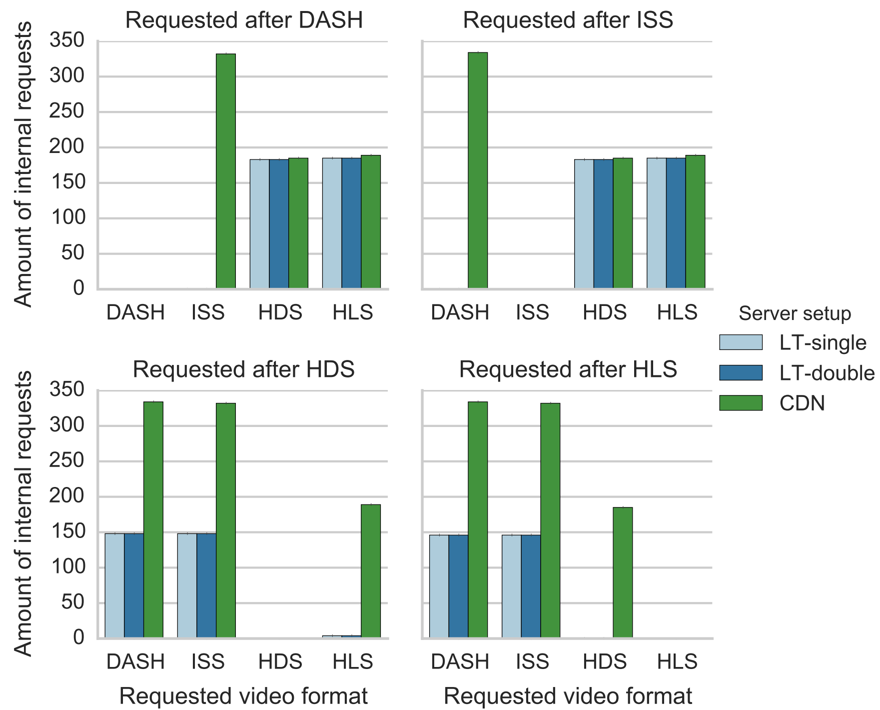

Figure 5.18: Amount of internal requests with the cache filled with another format

CHAPTER 6

Discussion

This chapter tries to explain and relate the many results gathered from the experiments described in the previous chapter. It also suggests opportunities for future research subjects.

6.1 Explanation of the results

Some results can be explained easily, some others need some more work. This is why the full explanation is split into smaller sections, which explain the results in an easy order.

6.1.1 DASH and ISS versus HLS and HDS

A red line through all the measurements is that HLS and HDS perform quite similarly and DASH and ISS do this as well. Even when all of the four formats perform quite alike, the pairs of formats perform even more alike. This similarity or difference can easily be explained by the type of files the different formats use. DASH and ISS use separate files for audio and video, while HLS and HDS combine those two. This means only about half the amount of files is needed when streaming HLS or HDS when compared to DASH or ISS, see section 5.1.3 for more information.

Because of this difference in amount of requests, comparing the different formats using handled requests per second is unfair. A good example of this can be found by looking at the plots for handled requests per second and received MB/s. While all the different formats perform quite similarly when looking at received MB/s, the formats HLS and HDS perform about half as good as DASH and ISS when looking at requests per second. This can only mean that the requests for HDS and HLS are, on average, about twice as large as the requests for DASH and ISS. Which in turn can be easily explained by separate requests for audio, since audio files are much smaller than video files and would thus lower the average request size when requested separately. Comparing the formats base on latency is also unfair, since HLS and HDS have to do more work for a request it is not really fair to compare them against DASH and ISS. While the comparing measurements for requests per second and latency is clearly not fair when comparing different formats, it can still be very useful to compare the different setups.

6.1.2 Internal traffic with a cold cache

Findings that are easily explained are the identical internal traffic statistics for pairs of setups when using a cold cache. The reason that the different CDN type setups are the same in this regard is caused by the cold cache. Every time the CDN setup requests a segment from the storage this is cached, so it can be used by a future request that is exactly the same. However, since every segment is only requested once, the CDN setup only builds the cache, but it is not able to use the stored data yet. This is why internal traffic statistics should not be influenced by this caching. The same holds true when comparing the LT-single and LT-double setups, since all that differs between those setups is that the LT-double setup caches its response to the client, which is again different for every request. As a last pair, the IPP and the LT-nocache setup

perform the same as well. This is most likely caused by the fact that these setups are almost identical, except that one runs on Nginx directly and the other on Apache, which is proxied through Nginx.

The more interesting information lies in the fact that the LT-nocache setup does not perform the same as the other LT setups, in contrast to the CDN and the CDN-nocache setup. This means that the subrequest caching employed by the two caching LT setups already has an advantage, which means that cache hits occur even when the clients requests are all different. This means that some of the range requests that get generated by the IPP directive are the same for different segments. By looking at the server log when requesting the ISS format, it quickly becomes clear, at least for ISS, which requests are happening multiple times. A couple of requests are done very often when requesting the video, see appendix A for the log. It shows that the video manifest is requested very often, as well as the start and end of the video and audio files. All of these data segments, the manifest and file headers and footers, contain information about how to play the video and audio files, things like bitrate and segment endings. This is why it is very probable that these requests contain data that is necessary for the USP software to generate the ISS response and that something similar is the case for the other formats.

What is also interesting, is that while the differences in requests are quite large between the different pairs, the differences in sent MB are quite small. There is no difference between the CDN setups and the caching LT setups, this is likely caused by the fact that both have to request the entire video from the storage. When comparing the IPP and LT-single setups to the other setups, there is a large difference in amount of requests, but a relatively small difference in amount of data. This indicates that apart from the actual video data needed to fulfill the requests, a large amount of small requests is sent as well. Which can be easily explained by the findings from the previous paragraph, since the requests that were done multiple times, were all not very large.

6.1.3 Cache usage with a cold cache

When measuring the cache usage with a cold cache, the different usages can be quite easily explained. First of all, the setups that have no caching enabled use almost no storage for cache, but they still use some. This is probably caused by Nginx its cache implementation, it uses directory layers to store the entries [30]. These directories itself also take up some space, even when they are empty. Secondly the LT-single and the CDN setup use about the same amount of cache, because they both store the entire video. They do store it in a different format however, the LT-single setup stores it in byte ranges of the original video file and CDN setup stores it in converted segments. Apparently this difference in format does not change the size much. Lastly, the LT-double setup uses about double the amount of cache that the LT-single and the CDN setup use. This is simply because it caches the video in both of the formats that these setups cache them in, which easily explains the increase in cache usage.

6.1.4 Transfer speed and latency with a cold cache

One important observation from the transfer speed plots is that all the different setups reach a maximum. This maximum is at about 12 MB/s for the caching LT and the CDN setups, and it is at about 10 MB/s for the IPP and LT-nocache setup. These to maxima are mostly caused by the bandwidth restriction of 100 Mbit/s on internal traffic. This can easily be shown for the first set of setups by showing that the maximum is almost equal to the bandwidth limitation, $100\,\text{Mbit/s} = 12.5\,\text{MB/s} \approx 12\,\text{MB/s}$. The lower maximum shown for the other set of setups can than be explained by the higher amount of internal traffic for those setups, shown in figure 5.5.

The difference in transfer speed between the different setups can mostly be explained by looking at the internal traffic statistics. The CDN type setups both sent the least internal requests and the least data, which is why they perform best. The caching LT setups sent the same amount of data, but they sent more requests, which means more overhead. This overhead is apparently most noticeable when doing a small amount of concurrent requests. Lastly the IPP and LT-nocache setups probably performed worst because they sent both the most amount of internal requests and the most data.

One thing that is not explained by the internal traffic is why the IPP setup performs worse, both in transfer speed and latency, than the LT-nocache setup when using more than one request. Since they are basically the same, there normally should not be such a large difference. One thing that could cause the slow down is the blocking nature of the IPP directive, described in the last paragraph of section 4.1. By removing the blocking code from Nginx, although originally meant to remove a deadlock, this problem can be avoided. Something else that is not explained is the difference in performance between the two caching LT setups. This is also a phenomenon in some of the other plots and that is why it is discussed later in section 6.1.7.

6.1.5 Results from a cache filled with the same format

The most easy results to explain for the tests where the cache is filled with the same format are the cache usage and the internal traffic results. No internal traffic has taken place for any of the caching setups, because the cache already contained all the data needed to fulfil the requests for the video segments. The LT-double and the CDN setup already have a cache that contains the exact responses that they have send to the client and the LT-single setup already has a cache filled with the byte ranges it needs to generate the requested segment

The reason why all caching setups perform better, transfer speed and latency wise, with their cache filled is also quite clear. No time is spent on internal traffic, so any delay that was caused there, when using a cold cache, is now gone. This is the case for all three caching setups. For the LT-double setup and the CDN setup, another phase is skipped as well, because of the caching. This phase is the video conversion. All these two setups have to do is get the cache entry and send it back to the client. The LT-single setup still needs to do the conversion phase, it has to get the range requests from cache and transform them into a segment. This is probably why the CDN and LT-double setup perform very similarly and why the LT-single setup performs better than with a cold cache, but not nearly as good as the other two setups.

The one thing that has not been explained yet is the drop in performance for the LT-double and CDN setup when adding more connections. The reason for this drop most likely has something to do with overloading the server. As mentioned before, the servers used in the tests are not very powerful and it is quite likely that the cache server could not handle the load of so many concurrent connections that all finished so quickly.

6.1.6 Results from a cache filled with another format

For the results where the cache is filled with another format than requested, data is different depending on which format is requested and which format was already in the cache. To make it easier to specify what was in the cache and what was requested, the same terminology will be used that was described in the previous chapter. To reiterate, when a line says that DASH was requested after ISS, it means that ISS was already in the cache and DASH was requested. This order of requests will be called a requesting order.

When looking at the results for this cache condition, the internal traffic measurements explain most of the other results as well. Which is why the internal traffic measurements will be explained first. The most obvious part of these measurements are the measurements for the CDN setup. These measurements are identical to the results obtained when testing with a cold cache. The reason for this is that the CDN setup has no use for the segments it has in cache, since segments in another format are requested and the ones in cache do not match them. This means that it still has to ask the storage server for all the segments. This is probably also the reason why the measurements for the two LT setups are the same. The only difference between the two is that the LT-double setup caches the responses it sends to the client. However, in this case those cached responses should have no use, since another format is requested.

The more interesting fact about the internal traffic measurements, is that the LT setups generate a lot less internal traffic then they did when running with a cold cache. All of the different requesting orders generate less internal requests and request less data. There are requesting orders where no requests are sent at all, which means that all the data necessary for the generation of the fragments was already in the cache. This is the case when requesting DASH after ISS, ISS after DASH and HDS after HLS. It is also almost the case when requesting HLS after HDS. In

that case however, a very small amount of requests still needs to be sent. All the other requesting orders still need some data to be sent, although it is a relatively small amount compared to the data the CDN setup needs. However, this small amount of data accounts for a large amount of requests, about as much as the CDN setup sends.

The most probable reason for this is that these extra requests are caused by the file differences mentioned in section 5.1.3. Both DASH and ISS need to download 147 audio files and HDS and HLS need to download 184 combined audio/video files. Because of this it is very likely that different audio ranges are requested by the IsmProxyPass directive, because the full audio file is split in a different amount of segments. These different requested ranges should cause cache misses, since they are not exactly the same as previous ranges, and that in turn means they have to be requested from the storage. This same argument could be made for the video segments, were it not the case that the amount of video and combined audio/video files are exactly the same for all formats, namely 184. Which is why the video ranges needed for both would match each other. The numbers seem to match up when looking at the amount of internal requests that still get sent, about 150 when requesting DASH and ISS and about 180 for HDS and HLS. This would also explain why the amount of data sent internally is so small, compared to the amount of requests, since audio is pretty small when compared to video.

The one thing that is not explained by this is, is the very small amount of requests that needs to be sent when requesting HLS after HDS. It is not directly clear what causes this. One thing that could be the cause is the large amount of playlist files HLS uses. It could very well be the case that one or more of those only need a small range of the video file, which does not match the ranges needed to generate the playlist for HDS.

When looking at the cache usage it is seen that the LT-single setup uses almost no more than it did after running on a cold cache. This can easily be explained by the fact that the amount of data sent internally was very little and that is the only thing that the LT-single setup caches. The CDN setup uses about double as much as it did with a cold cache, this can also easily be explained by the fact that it sent about the same amount of internal data, which is also cached again. The LT-double setup is increased by about half and this can again be explained by the fact that it caches all the data that the other two setups cache, which put together is about half as much.

When looking at the transfer rate and latency data it comes as no surprise that the CDN setup performs about the same as it did with a cold cache, since it should essentially do the same. What is very interesting though, is the performance of the two LT setups. It is both much better than the performance of the CDN and that of the setups when using a cold cache. These large differences can only be explained by the fact that so little data has to be sent internally. The difference is largest when multiple concurrent connections are used. The LT-single setup, with enough connections, performs about five times better than the CDN setup for the requesting orders where (almost) no requests had to be sent internally. The LT-double setup even performs about seven times better than the CDN setup under these conditions. The LT setups even perform much better than the CDN setup for the requesting orders where some data had to be sent internally, about four times better with the LT-single setup and five times with the LT-double setup. The latency measured is also much lower for the LT setups than for the CDN setup in all the requesting orders.

6.1.7 The speed and latency improvement of double caching over single caching

The one thing that has not been explained yet is the difference in transfer speed and latency between the two caching LT setups. The positive effects of the second cache in the LT-double setup have a clear explanation when the cache is filled with the same format as requested, which was explained in section 6.1.5. However, both with an empty cache and a cache filled with another format the LT-double setup clearly outperforms the LT-single setup. In these cases this performance gain cannot be explained by getting actual cache hits in its extra cache, since none of the clients requests match earlier ones. The only explanation left is that the actual caching of the responses has a beneficial side effect for these performance measures. It could be that Nginx does something clever, that makes proxying responses faster when they are allowed to be saved to disk. It is very unclear however, what exactly causes this weird but beneficial behaviour.

6.2 Future work

Some of the results and findings in this thesis present possibilities for future research. The first thing that begs for further investigation are the unexpected performance differences between the two caching LT setups. There occurred large performance gains just when caching responses, apart from the expected gains when being able to serve from the cache Two other options for future research are based on the shortcomings in existing technologies that were found when creating an implementation of the LT setup. The first of these is improving the current suboptimal methods of caching range requests. For instance by creating something similar to the theoretical solution described in section 3.3. Something else that could be improved in future research would be the IsmProxyPass directive. In its current blocking state it limits Nginx its performance and it would be interesting to see that changed. This would mean no second HTTP server would have to run on the proxy server to implement the LT setup. A last and logical direction for future research is investigating the performance of the two caching LT setups at a large scale, with lots of servers and lots of different videos. It could very well be that there are some issues that only occur when deploying at a large scale, which could require some extra tweaking of the setups.

CHAPTER 7

Conclusion

Streaming video has become a large part of the internet. Currently a well known server setup for streaming video is a content delivery network (CDN), which caches popular videos for quick access. One of the problems with this approach is that it requires caching of the videos in every streaming format that is supported. This thesis investigated the possibility of caching the original video file instead of the different formats, which could then be converted to each format when requested. To accomplish this, two different setups have been developed, one which caches only the original video file and another, which also caches the different formats after conversion has taken place.

To compare the different setups, their performance was tested under three different cache conditions. For the first condition the cache was simply left empty. For the second condition the cache was filled before the actual test started by requesting the same format that would be requested during the test. The last cache condition was almost the same as the second, but instead of filling the cache in advance by requesting the same format, one of the other formats was requested. Both new setups performed a little bit worse than, or equal to, the CDN setup, when the cache was cold. However, when the cache was filled with a different format they outperformed the CDN setup in speed and latency by as much as four to seven times. The internal traffic between the storage server and the proxy server is also a lot lower in this case and the new setup that only caches the original video file, used about half as much storage for cache as the CDN setup. Finally, during the tests where the cache was filled with the same format, both setups perform better than they did with an empty cache. However, only the new setup that also caches the formats after conversion performs, speed and latency wise, on par with the CDN setup. A last important difference between the new setups and the CDN setup is where they convert the original video files. The CDN setup requires the storage server to convert the original video file to the different formats. The new setups convert the video on the proxy servers. The advantage of doing the conversion on the proxy servers is that the video storage can be handled by cheap dedicated storage services that only provide file hosting and no processing power, such as Amazon S3.

All in all, the new setups are valuable additions to currently known setups. The new setup that caches most serves content faster than, or as fast as, the CDN setup and it also generates a lot less internal traffic. The only negative aspect is that this setup uses more storage for cache than the CDN setup. The version that caches only the original video file still has the internal traffic advantages that the other setup has, but it performs worse than the CDN in speed, when requesting the same format multiple times. However, it can serve multiple formats using half the amount of cache the CDN setup uses. This means that depending on the main requirement, speed or cache usage, one of the new setups can be chosen to outperform the traditional CDN setup in this aspect.

Bibliography

[1] T. D. Monchamp, “Adaptive bit-rate streaming”, *InSight: Rivier Academic Journal*, vol. 9, no. 2, 2013.

[2] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee, *Hypertext transfer protocol–HTTP/1.1*, 1999.

[3] *Transmuxing*. [Online]. Available: `https://app.zencoder.com/docs/api/encoding/transmuxing` (visited on 17/06/2015).

[4] A. Zambelli, “IIS smooth streaming technical overview”, *Microsoft Corporation*, vol. 3, 2009.

[5] Microsoft Corporation, “[MS-SSTR]: smooth streaming protocol”, 2014.

[6] Adobe Systems Incorporated, “HTTP dynamic streaming specification version 3.0 final”, 2013.

[7] R. Pantos, “HTTP live streaming”, 2015.

[8] T. Stockhammer, “Dynamic adaptive streaming over HTTP–: standards and design principles”, in *Proceedings of the second annual ACM conference on Multimedia systems*, ACM, 2011, pp. 133–144.

[9] International Organization for Standardization, “Information technology – dynamic adaptive streaming over HTTP (DASH)”, 2014.

[10] A. Vakali and G. Pallis, “Content delivery networks: Status and trends”, *Internet Computing, IEEE*, vol. 7, no. 6, pp. 68–74, 2003.

[11] A. Stricek, “A reverse proxy is a proxy by any other name”, *SANS Institute InfoSec Reading Room*, vol. 13, 2002.

[12] R. Fielding, Y. Lafon and J. Reschke, “Hypertext transfer protocol (HTTP/1.1): Range Requests”, Tech. Rep., 2014.

[13] *HTTP proxy*. [Online]. Available: `http://docs.unified-streaming.com/documentation/vod/http-proxy.html` (visited on 10/06/2015).

[14] C. Nedelcu, *Nginx* HTTP *Server: Adopt Nginx for Your Web Applications to Make the Most of Your Infrastructure and Serve Pages Faster Than Ever*. Packt Publishing Ltd, 2010.

[15] W. Reese, “Nginx: the high-performance web server and reverse proxy”, *Linux Journal*, vol. 2008, no. 173, p. 2, 2008.

[16] J. Ellingwood. (2015). Apache vs Nginx: practical considerations, [Online]. Available: `https://www.digitalocean.com/community/tutorials/apache-vs-nginx-practical-considerations` (visited on 10/06/2015).

[17] O. Garrett. (2015). Inside NGINX: how we designed for performance & scale, [Online]. Available: `http://nginx.com/blog/inside-nginx-how-we-designed-for-performance-scale/` (visited on 11/06/2015).

[18] *Apache*. [Online]. Available: `http://docs.unified-streaming.com/installation/advanced/stream/apache.html` (visited on 10/06/2015).

[19] *Nginx*. [Online]. Available: http://docs.unified-streaming.com/installation/advanced/stream/nginx.html (visited on 10/06/2015).

[20] *Module ngx_http_proxy_module*. [Online]. Available: http://nginx.org/en/docs/http/ngx%5C_http%5C_proxy%5C_module.html#proxy%5C_cache (visited on 28/05/2015).

[21] *Apache module mod_cache*. [Online]. Available: %7Bhttps://httpd.apache.org/docs/2.2/mod/mod%5C_cache.html%7D (visited on 10/06/2015).

[22] *HttpLuaModule*. [Online]. Available: http://wiki.nginx.org/HttpLuaModule (visited on 28/05/2015).

[23] M. Tourne. (2012). Pushing nginx to its limit with lua, [Online]. Available: https://blog.cloudflare.com/pushing-nginx-to-its-limit-with-lua/ (visited on 10/06/2015).

[24] *Apache module mod_lua*. [Online]. Available: https://httpd.apache.org/docs/trunk/mod/mod%5C_lua.html (visited on 11/06/2015).

[25] M. Dounin. (2010). Re: reverse proxy + caching seems to break range requests, [Online]. Available: http://forum.nginx.org/read.php?2,71160,71224#msg-71224 (visited on 28/05/2015).

[26] C. Aquino. (2014). Caching range requests using NGINX at MaxCDN, [Online]. Available: http://syshero.org/post/77122862845/caching-range-requests-using-nginx-at-maxcdn (visited on 24/04/2015).

[27] T. Treat. (2015). Comcast, [Online]. Available: https://github.com/tylertreat/Comcast/blob/master/README.md (visited on 12/06/2015).

[28] W. Glozer. (2015). Wrk - a HTTP benchmarking tool, [Online]. Available: https://github.com/wg/wrk/blob/master/README (visited on 12/06/2015).

[29] *Unified Capture*. [Online]. Available: http://docs.unified-streaming.com/documentation/capture/index.html (visited on 17/06/2015).

[30] *Module ngx_http_proxy_module*. [Online]. Available: http://nginx.org/en/docs/http/ngx%5C_http%5C_proxy%5C_module.html#proxy%5C_cache%5C_path (visited on 15/06/2015).

Acronyms

ABS adaptive bitrate streaming. 5, 7, 20

CDN content delivery network. 6, 8–10, 12, 13, 19–21, 23, 25–27, 29, 31–33, 35–38, 41

DASH Dynamic Adaptive Streaming over HTTP. 6, 20, 22, 25, 26, 29–33, 35, 37, 38

HDS HTTP Dynamic Streaming. 5, 20, 22, 25, 26, 29–33, 35, 37, 38

HLS HTTP Live Streaming. 6, 20, 22, 25, 26, 29–33, 35, 37, 38

IPP IsmProxyPass. 9, 10, 12, 13, 15, 17, 19–21, 23, 25, 35–39, 47

ISS ISS Smooth Streaming. 5, 20, 22, 25, 26, 29–33, 35–38

LT late transmuxing. 15, 19–21, 23, 25–27, 29–33, 35–39

USP Unified Streaming. 5, 7, 9–12, 15, 19, 20, 36

APPENDIX A

Range requests done for the ISS format

This is a list of the range requests that are generated by the IsmProxyPass directive when requesting the video used during testing. The first column contains the amount of times the specific request was generated and the second column describes the request, showing first the URL and then the range. It only shows the top 20 requests, since there are a lot of requests that only occur once, which are not very interesting and would take up a lot of space.

```
332 "GET /video/tears-of-steel/fmp4/tears-of-steel.ism HTTP/1.1" [-]
184 "GET /video/tears-of-steel/fmp4/tears-of-steel-1200k-bp.ismv HTTP/1.1" [bytes=112729239-112794774]
184 "GET /video/tears-of-steel/fmp4/tears-of-steel-1200k-bp.ismv HTTP/1.1" [bytes=0-4095]
148 "GET /video/tears-of-steel/fmp4/tears-of-steel-128k.isma HTTP/1.1" [bytes=16983917-17049452]
148 "GET /video/tears-of-steel/fmp4/tears-of-steel-128k.isma HTTP/1.1" [bytes=0-4095]
  1 "GET /video/tears-of-steel/fmp4/tears-of-steel-400k-bp.ismv HTTP/1.1" [bytes=37367189-37432724]
  1 "GET /video/tears-of-steel/fmp4/tears-of-steel-400k-bp.ismv HTTP/1.1" [bytes=0-4095]
  1 "GET /video/tears-of-steel/fmp4/tears-of-steel-128k.isma HTTP/1.1" [bytes=9989376-10100427]
  1 "GET /video/tears-of-steel/fmp4/tears-of-steel-128k.isma HTTP/1.1" [bytes=9987520-9989567]
  1 "GET /video/tears-of-steel/fmp4/tears-of-steel-128k.isma HTTP/1.1" [bytes=9875258-9987519]
  1 "GET /video/tears-of-steel/fmp4/tears-of-steel-128k.isma HTTP/1.1" [bytes=9873402-9875449]
  1 "GET /video/tears-of-steel/fmp4/tears-of-steel-128k.isma HTTP/1.1" [bytes=9760133-9873401]
  1 "GET /video/tears-of-steel/fmp4/tears-of-steel-128k.isma HTTP/1.1" [bytes=9758277-9760324]
  1 "GET /video/tears-of-steel/fmp4/tears-of-steel-128k.isma HTTP/1.1" [bytes=9643585-9758276]
  1 "GET /video/tears-of-steel/fmp4/tears-of-steel-128k.isma HTTP/1.1" [bytes=9641729-9643776]
  1 "GET /video/tears-of-steel/fmp4/tears-of-steel-128k.isma HTTP/1.1" [bytes=9527980-9641728]
  1 "GET /video/tears-of-steel/fmp4/tears-of-steel-128k.isma HTTP/1.1" [bytes=9526124-9528171]
  1 "GET /video/tears-of-steel/fmp4/tears-of-steel-128k.isma HTTP/1.1" [bytes=948043-1063772]
  1 "GET /video/tears-of-steel/fmp4/tears-of-steel-128k.isma HTTP/1.1" [bytes=946187-948234]
  1 "GET /video/tears-of-steel/fmp4/tears-of-steel-128k.isma HTTP/1.1" [bytes=9411579-9526123]
    ...
    ...
    ...
```