FileSender Terabyte Challenge

Research Project 1 - Report

René Klomp
Rene.Klomp@os3.nl

Edwin Schaap
Edwin.Schaap@os3.nl

February 11, 2013

The purpose of this research project is to identify and solve the bottlenecks in the FileSender application. Besides that it will also look into some more generic problems that can affect more web applications. The main problem in FileSender is that there are gaps between sending chunks. This report will describe this problem and show our implementation for solving this using JavaScript's multi-threading support called webworkers. It will also discuss SSL performance using different ciphers and give some guidelines for choosing the best performing cipher.

1. Introduction

The FileSender project [4] has issued the Terabyte Challenge [10]. Current upload speeds with FileSender are nice for files up to several GBs. We want to enable use of FileSender to transfer a 1 TB file in a reasonable amount of time (5 hours on a low latency path) using a standard web browser on a standard Windows, Linux or Mac computer. Our goal is to identify current performance bottlenecks and design possible solution strategies which hold as latency increases. The main question that this research is based on is:

> *Can we identify bottlenecks in the FileSender application and how can we improve the transfer speeds by reducing or removing these bottlenecks.*

1.1. Problem

With a default setup of FileSender uploading a large file is not going as fast as it is expected to be. While our setup (section 1.4) contains equipment to go up to 1 Gbit/s, the file upload speed did not exceed 160 Mbit/s.

This observation shows that there is clearly something within the setup that restricts the ability to use the full bandwidth that is available.

1.2. FileSender Constraints

The FileSender product is meant to be easy. In the first place it should be easy for the users. This means that users are not required to install any extra programs to be able to use FileSender and upload their files. The motto is that if a user can use YouTube he/she can use FileSender. The user can just browse to the FileSender webpage, login, select their file and forget it until it is finished uploading

Besides being easy for users, the goal of FileSender is also to be easy for system administrators. System administrators should be able to install a fully working version of FileSender within an hour. To achieve this, the FileSender backend is completely written in PHP. It has almost no dependencies on external libraries and uses the standard PHP functionality as much as possible. The frontend is written using standard HTML5 functions and JavaScript, and is served to the browser of the user. Therefore the user does not have to install any extra software. The PHP application only has to be extracted on any PHP supported webserver (with some small requirements) and after configuring the application it is directly usable by the user.

We have to keep the above described constraints in mind when creating a solution for the FileSender problem.

1.3. Methodology

For this research we went through several steps to obtain results:

Hypothesis
: Based on code review from the default installation we came up with a hypothesis. (section 2)

Experiment
: To see if the hypothesis stands, a first experiment was conducted with a lab setup (section 2). To exclude other causes, we have investigated the hardware (section 3) and software (section 4) from the lab setup.

Analyze
: In this step, results where analyzed from the experiment whether hypothesis is correct. Problems found in the hardware and software of the setup were also covered.

Solutions
: We have come up with solutions for the problems that are found (section 5) and created a prototype that removes the bottleneck (section 5.1).

Reflection
: To see if the prototype is valid, it is reflected on the FileSender application to verify results (section 5.1.4). And we subsequently subjected our implementation to the terabyte challenge (section 6).

1.4. Experimental environment

During this research we have used a Dell PowerEdge R420 server with a standard Ubuntu Server 12.10 installation to run the FileSender PHP application on. The server is equipped with two Xeon E5-2430L processors and 32GB of RAM. For storage 6 hard-disks, with a capacity of 500GB each, were used in RAID5 which gave us roughly 2.5TB of storage.

As client we have used two Dell Latitude E6220 laptops of which one was equipped with a regular hard disk and the second with a SSD. Both laptops had a default installation of Elementary OS [3], a variant on Ubuntu 12.04, and the disks were partitioned with the ext4 file-system [5].

We had a 1Gbps low latency (<1ms) network link available to connect to the server using our laptops and we have used NetEm [22], a network emulation tool, to simulate latency. For most tests Google Chrome [7] was used as the browser.

A full list of technical details about the setup can be found in appendix C.

All our tests where conducted using random data files which where generated from `/dev/urandom`. This means that our files may contain less entropy than when we had used `/dev/random`, but for the sake of our experiments this is good enough. We did not use files filled with zeros as these can easily be compressed and can cause unexpected results during test. The reason for this is that browsers and server can use compression in the HTTP bodies. To eliminate this factor, random data was used.

To keep this report consistent all results will be given in bits per second unless explicitly mentioned otherwise.

2. First Hypothesis

The first thing we noticed when looking at the code, is that FileSender is using chunks to upload files. It iteratively walks through the file and takes chunks of equivalent size (except for the last one). When it reads a chunk from the file the chunk is subsequently uploaded to the server and an event handler is created to handle the response of the server. At the moment this event is fired, and thus the server has finished processing the file, and responded to the client, the client continues by sending the next chunk. In the meantime, between sending the last bit of a chunk and having received the full reply of the server, the client is idle, and thus not utilizing all available bandwidth.

To see if this hypothesis stands, a test was performed by uploading a file to a standard installation. During the upload process, a snapshot was made of the network traffic with Wireshark [11] and visually analyzed to give some clearance about what happens.

The problem is clearly visible in figure 1. This plot is a very small snapshot of an upload session. After each chunk a gap is created because the client is waiting for a reply. As latency increases this gap also gets bigger. The round trip time is added to the serverside processing time for each chunk.

If we translate this to a complete file upload of 10GB, this will take just over 18 minutes with a RTT<1ms. If we simulate the RTT from Utrecht to Washington DC and back, which is around 100ms, the upload took 31 minutes. Since the 100ms are added within the gaps, this alone will result in 9 minutes delay because of the 2MB chunksize. So, in conclusion to this observation, the higher the delay of the network, the greater the gaps will become.

In section 5.1 we will discuss our implemented solution to remove these gaps by using so called webworkers, the multi-threading support in JavaScript and HTML5. In addition, in section 5.4.1 we will discuss whether different chunk sizes impact performance.

3. Determining whether hardware is the bottleneck

In the previous section we have shown that the gaps introduced in the file uploads indeed can cause the problems described in section 1.1. In this section we will systematically benchmark all components of the systems, on the path from the client to the server, that can affect the upload performance to exclude that any of those hardware parts can form another bottleneck.

We will first start in sections 3.1 and 3.2 with determining the hardware limits on client and server. Then in section 3.3 we will determine the maximum network speed between the latter two.

The complete test results of all benchmark tests discussed in the following sections can be found in appendix A.

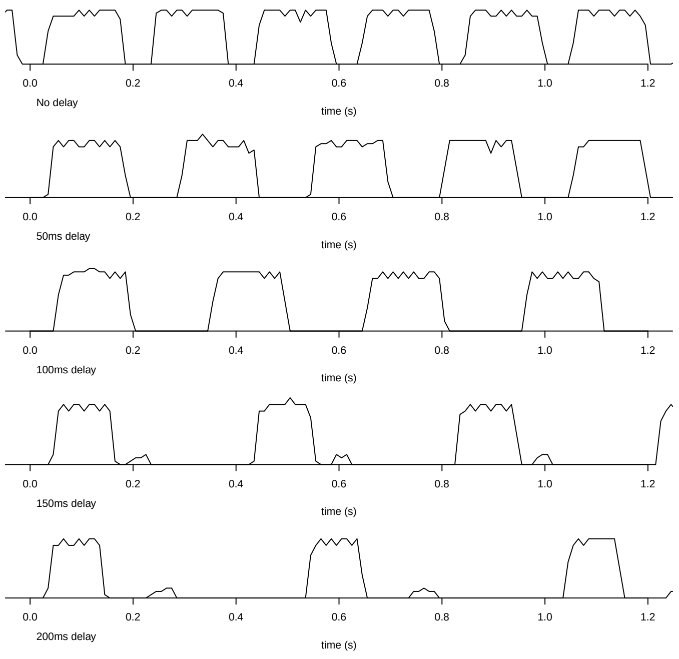

Figure 1: *Gaps while transferring chunks. Latency increase from top to bottom: 0ms, 50ms, 100ms, 150ms and 200ms.*

3.1. Client

3.1.1. Hard Disk

When uploading a file the file is first read from disk in chunks. The maximum read speed on the two test laptops have been measured using the `dd` program. To clean the caches to make sure the file is fully read from disk and not from memory we use:
`echo 3 > /proc/sys/vm/drop_caches`
This will cause the kernel to free *pagecache*, *dentries* and *inodes* from memory, causing that memory to become free. After making sure all caches are empty the following command will read data from disk and write it to `/dev/null`:
`dd if=randomdata.bin of=/dev/null`.
When completed, the command will output the average read speed.

Results For both laptops the above test was repeated 5 times. On the laptop with the SSD the average read speed was measured at 3742.40 Mb/s with a standard deviation of 56.68. The laptop with the regular spinning disk hard drive gave over all tests an average of 825.60 Mb/s with a standard deviation of 11.87.

3.1.2. Processor and Memory

To check whether the processor or memory could be the bottleneck we simply started an upload to FileSender. During the upload we monitored the processor and memory usage and without any doubt they are not the bottleneck. The processor was being used more than when the laptop was idle, but not one of the cores reached more than 50 percent.

3.2. Server

3.2.1. Hard Disk

As with the read speed on the client the write speed on the server has been measured with the dd program. The command used here was:

```
dd count=1M bs=60k if=/dev/zero
of=/tmp/test.img
```

We can safely use /dev/zero here, because no compression will be used when writing to disk. Besides, if we would have used /dev/urandom this would have generated so many CPU cycles that the processor would have been the bottleneck.

After 5 testes this gave an average maximum write speed of 4118.40 Mb/s with a standard deviation of 186.54.

3.2.2. Processor and Memory

Like with the client, to test the memory and processor usage on the server we monitored both during a file upload. We observed that only a single core (of the 24 cores) was used by the server and this core was only used for 60 percent. In addition to this, no extreme memory usage was seen.

3.3. Network

Now we know what our server and client are capable of, the last thing to benchmark is the network link between them. This was tested using iperf [30], a tool that estimates not the available bandwidth, but the achievable TCP throughput. The TCP throughput is the throughput that is relevant because uploading to the server goes through TCP and this gives some overhead and an increased size from the tcp headers. For this possible bottleneck we did 5 measurements in both directions by letting iperf run for 50 seconds and then getting the average throughput every 10 seconds. As expected the throughput from client to server and vice versa is almost 1 Gbit/s. The average throughput is from client to server 935.40 Mb/s and from server to client 941.60 Mb/s both with a very small standard deviation.

3.4. Hardware bottleneck conclusions

From the tests discussed in the preceding sections we can conclude that neither the hardware on the client side nor the hardware on the server side is a limiting factor for the throughput. With one exception, namely the read speed on the client with the regular hard drive. But still, this is not the bottleneck that produces the problems as described in section 1.1. Given that we use the client with the solid state disk for transfer speeds higher than the regular hard disk's read limit of 825 Mb/s, our theoretical maximum throughput speed is limited by the maximum network throughput which is around 935 Mb/s.

4. Determining whether software is the bottleneck

In the previous section we tested all parts of the hardware and concluded that this is not the bottleneck that is causing the low throughput as found in section 1.1. Based on this, one would expect that the problem is caused by the software itself. In section 2 we already showed the gaps, so the next sections we will discuss more general problems that can be caused by the software needed for FileSender.

4.1. The Browser

Since the FileSender user will be using only a browser to access the application this also needs to be evaluated. For the experimental setup we will be using Google Chrome [7] mostly as described in section 1.4. To exclude the problem of this browser we tested also FireFox [8].

For the evaluation of the two browsers the performance of the client system was observed. When using the FileSender application no significant performance issues were found for Google Chrome and FireFox. The CPU showed some activity but no more than any other interactive website. The memory usage was not an issue with all browsers and was stable when using the application.

4.2. Webserver

Other important software can be found at the serverside of the application. The PHP application is served with a webserver and therefore must be evaluated if this causes performance issues. The default FileSender setup was installed with Apache webserver [2]. We executed an installation with Nginx [9] to see if the Apache webserver is a bottleneck. The same FileSender installation was served by both webservers on different ports. We tested both with and without a secure connection. For the insecure connection there where no differences noticed in upload speed. For the secure connection there was a negligible difference noticed in the favor of Nginx. We noticed that the SSL caused a slower upload speed than the insecure connection, on both webservers. In the next section we dive further into this.

Now that we do not see a real difference between the webservers we need to test if the webserver is in any case capable of handling high network speeds. We performed an upload test with Apache Benchmark [1] to send chunks of data to the server. To test the download speed we used Wget [6] to download a static file from the server. As a result, the upload speed of an insecure connection was found to be 880.96 Mb/sec and for the secure connection 537.44 Mb/s. For the download speed the results for both types of connection were around 900 Mb/s. When testing with the default browser in our setup, we got the same download speed over the insecure network connection but only around 200 Mb/s over the secure connection. So we have seen that the webserver can handle high speeds, both upload and download, over an insecure connection. The secure connection causes some confusion so in the next section we will investigate this.

4.3. SSL

When testing what our maximum download speed from the server to one of the clients would be we discovered a very notable speed difference between Google Chrome and GNU Wget [6]. While Wget was reaching our maximum possible download speed with approximately 900 Mb/s, Google Chrome only reached around 200 Mb/s. When analyzing the SSL handshake packets, it turned out that Google Chrome used the Camellia [14] algorithm for encrypting the SSL session while Wget used the more known AES (advanced encryption standard) [15]. Both programs where using a 256 bits key. Although they are two different ciphers, Camellia offers comparable encryption speed [14]. But this should not give the four times slower result. The reason for this can be explained by the fact that the processor in the laptops are supporting Intel's AES-NI instruction set [17]. This instruction set provides hardware support for the AES cipher and should make it a lot faster.

To prove this we ran OpenSSL speed tests for both algorithms. First we ran the Camellia and AES test with OpenSSL without AES-NI support. Then AES was tested again with OpenSSL with AES-NI support. This was conducted 5 times for each test. The average results of 5 measurements can be found in table 1 and show a clear improvement in speed. If we do not have AES-NI support we can see that the Camellia cipher is somewhat faster (almost 1.6x) than AES. But when AES-NI is available, this cipher is almost 6 times faster and 3.8 times faster than Camellia.

In section 5.4.2 we will discuss and compare other encryption ciphers and look deeper in this problem and what the best solution will be for the SSL bottleneck problem in FileSender and other network intensive web applications.

	Speed Mb/s
Camellia	974,36
AES	628,26
AES-NI	3.725,23

Table 1: *Comparing Camellia against AES with and without AES-NI support*

4.4. TCP

The last problem we want to discuss is the problem of the so called TCP bandwidth-delay product [24, 27]. A decrease in transfer speed is caused by the high latency between two endpoints.

On a 1Gbit connection line with an RTT of 500ms, there can be approximately 63 MBytes of data in transit. The default TCP window is often 65535 bytes without scaling and thus the full bandwidth capacity cannot be reached. When using autoscaling the maximum window size can be 2^{30} bytes [23]. But since the TCP default settings can sometimes not be appropriate for Long Fat Pipe, one can use TCP Tuning to optimize these.

Since we only focused on a low latency path, our setup (section 1.4) has a latency of less than 1 ms, TCP tuning is not applicable to our setup. To really be able to test this we would have needed more time, which we simply did not have. Besides this, given the FileSender constraints it would probably not be a realistic possibility since using JavaScript we cannot modify any TCP settings. Therefore we will consider TCP tuning as future work.

4.5. Software bottleneck conclusions

In this section, we have seen that the different browsers are capable of handling the upload. While uploading a file, all browsers did remain responsive and were capable of handling even more.

The server software was also able to handle full bandwidth speed and there was no difference between Apache webserver and Nginx. Finally we have looked at the secure connection with SSL between the client and server and this caused some trouble. Depending on the system and chosen cipher, this could be a bottleneck in the system. When the Camellia cipher was chosen or AES-NI was not available on the system, the maximum bandwidth could not be utilized.

In section 5.4.2 we will provide advise on how to deal with this bottleneck.

5. Proposed Solution

5.1. Improved implementation

In order to remove the gaps, we had to come up with a solution that can fully benefit from the available bandwidth. Since the problem in the application is that the client waits for a response before it proceeds to the next chunk, we can probably make this smarter. The first method is to send the next chunk directly after the previous one is send. This would improve the performance but small gaps would still exist. This is caused by the fact that the system would first read the chunk after which it would send it. During the read, no data will be send. Our hypotheses was that by using parallelism we could make sure that during those gaps another thread was still sending data and thus utilizing more bandwidth.

5.1.1. First Concept

To test whether this solution would even work, we started by creating a small proof of concept application. This application only contained some JavaScript and HTML code to send chunks parallel to the server. The server did not have code running to handle those packages so they where just being discarded.

Whenever a file is selected in a HTML file field, the JavaScript code starts a predefined number of workers and passes to these new workers two arguments. The first argument is the start byte where the worker has to start reading the file. The second is the chunks size so that the worker knows how much data to read. With this information the worker can now do its job and read the chunk from the selected file and subsequently upload this chunk to the server. When the chunk has been uploaded and a reply from the server has been received, the worker reports back to the main thread and the main thread will give the worker a new startbyte so it can send another chunk. This keeps repeating until the whole file is uploaded.

Our concept turned out to work as expected. By making use of parallelization the gaps where eliminated and full throughput speeds could be reached. The next challenge was to implement the server side as we will describe in the next section.

5.1.2. Server side out-of-order algorithm

Sending chunks to the server in parallel is not a very big challenge, but the problem with this approach is that a chunk can be received at the serverside faster or slower than another chunk. The big challenge arises when we have to deal with these out-of-order chunks at the server side. To solve this problem we had to devise an algorithm to append the chunks in the right order to the destination file.

The questions that had to be answered where:

- How do we know at the server side which chunk we are dealing with?
- When can we append a chunk and when do have have to queue it?
- How do we maintain our queue?
- How do we make sure that in the end all chunks are appended in the right order?
- What problems can arise from concurrency?

To know at the server side which chunk we are dealing with we choose for a simple solution. For every chunk the worker adds to the HTTP request a new header called `X-Start-Byte`. This header contains, as it name suggests, the start byte for that particular chunk. To know whether the chunk can be appended to the destination file, the server can now compare this header with the current file size. If the two match we know that the chunk is the next chunk that must be appended, so we can append it.

But what happens when a chunk arrives to early. That is, what happens when the `X-Start-Byte` header is bigger than the current file size. In this case we have to store our chunk in a queue, but how do we implement such a queue?

Our first toughs were to do this in the FileSender database. We could have stored our chunk in the database and later on just query the database to get our queue. When discussing this option we decided that our improved upload implementation should be a separate library so that it can be used for other projects as well. So to keep our library simple, and to stay in line with the FileSender philosophy of keeping it easy to install, we thought that using a database was not a good idea. This also means that we are designated to the filesystem for keeping our queue.

To implement this queue on the filesystem we came up with a simple and robust solution. Whenever a chunk has to be stored we simply create a md5 of the destination file name and append a `#` followed by the start byte of that particular chunk and store this file in the temp folder. To get the contents of our queue we can simply get a list of all files starting with `[md5 of dest]#`. We now have have a simple queuing mechanism that stores all information we need, namely the chunk data together with its start byte.

The last thing we had to solve was, when do we append those chunks from the queue? We can do this whenever a chunk is late. That is, it cannot be appended because the `X-Start-Byte` header is bigger than the current file size, *but* the chunks needed before we can append this chunk are available in the queue. We can then simply append those chunks first, making the filesize equal to `X-Start-Byte`, and thereafter append the chunk that was late.

If the client is done uploading, it will send an empty chunk to make sure that any chunks left in the queue will be appended to the destination file to make in complete. After this, the upload is finished. To prevent those problems, we have placed all operations on the destination file behind a lock. Because two chunks can arrive at the same time, concurrency problems can arise during this process.

A simple pseudo code to further illustrate this algorithm can be found in appendix D.

Why not use fseek? Another way of solving this problem is simply using fseek to place the chunks at the correct location in the destination file. But while our solution is a little complex with our queuing mechanism, we deliberately choose to not use fseek. It is possible in PHP to use it, but the drawback of using it is that we then still have to keep an administration on which chunks are received and which are not. In our setup we use the filesize to determine how far the upload is and to be able to resume it again. But when we use fseek, whenever we stop an upload and want to resume it later on, we need to know which chunks are received, and which are not.

5.1.3. Implement in FileSender

Our client side implementation based on our proof of concept (section 5.1.1) together with our server side implementation (section 5.1.2) form a standalone library [26] that can also be used with FileSender. This design allowed us to fully integrate the library into FileSender trunk within a short time-span of two hours [25].

5.1.4. Verify Implementation

We have extensively tested our implementation and we can conclude that it works. In section 5.4.1 we show that chunksizes still have some effect on upload performance, but in figure 3 we can see that we can read the maximum throughput speed on a low latency path. In section 6 we will show what our implementation is capable of.

We have also verified that our implementation works in Google Chrome, Firefox and Internet Explorer 10. Altough, we observed that Internet Explorer 10 used almost 90% of the CPU, while Chrome and Firefox reached a maximum of 60%.

Another and also big advantage of parallelism is that we can use multiple TCP stream in parallel. We will describe how this can improve performance and why it is an advantage in the next section.

Worst case scenario In our setup the worst case scenario will probably be that the first chunk is received after all other chunks have been received by the server. This causes that all other chunks are queued until that first chunk is received as well. When that chunk arrives we then can append all chunks from the queue and finish our file.

5.2. Multiple TCP sessions

As said in section 5.1.4 parallelism allows us to use multiple concurrent TCP streams. Parallel TCP streams have been widely used to increase transfer speeds. A good example is GridFTP [12], an extension to the standard File Transfer Protocol (FTP) and used by e.g. the Globus project. One of the features that GridFTP uses to improve the performance is parallel data transfer. A lot of research has been done about the advantages and problems with multiple TCP streams [13, 19–21], and also in relation with GridFTP [28].

The advantage of multiple TCP sessions is that each session will have its own send/receive buffers. This way we can take care of the huge bandwidth-delay product for higher latency links. Lu et al. [29] describe different scenarios and show that increasing the number of TCP sessions indeed increase the throughput. So based on their results we can argue that using parallel file transfers is an advantage over just removing the gaps between chunks.

We also discovered that Google Chrome and FireFox, by default, do not make more than 6 connections per host. Internet Explorer will go up to 10 connections per host. So in general, creating more than six workers will not have much effect.

5.3. Other file upload solutions

There are some existing JavaScript libraries which also support large file upload. One of them is Plupload. This library does support large files but uploads them only asynchronous. This way, the gaps are smaller but not completely removed. Because of this construction, only one TCP connection is used. GridFTP on the other hand uses multiple simultaneous connections. It can fully benefit of the available bandwidth. Because of the construction of GridFTP, the user is required to install additional software. Since this doesn't fit within the constraints of FileSender, this is not an option.

5.4. Finding the best configuration

As described in section 5.1.3, we now have an improved implementation for uploading files to the FileSender instance. In this section we will look deeper in the different configuration options available for FileSender and how those options can affect performance. In section 5.4.1 we will look at different chunk sizes. Does a bigger chunk give better performance? As we already mentioned in section 4.3, SSL can form a bottleneck in the system, so in section 5.4.2 we will discuss different SSL ciphers and which one can be used best.

5.4.1. Chunk Size

In this section we will look at different chunk sizes and how those impact performance. We have tested chunksizes ranging from 0.5 to 100 MByte on both SSL and non-SSL connections. The measurement is the average throughput on a 120 second interval on the network interface. So this is not the actual file throughput and not extremely accurate, but it gives a good insight in how the chunksize influences the upload speed.

Keep in mind that the results shown are results from a *lab setup*. A real world environment would probably have a slightly worse performance.

During these test the SSL connection was using AES encryption.

Figure 2 shows, for the original upload implementation, how the throughput speed is related to the chunks size. This figure clearly shows that for small chunksizes increasing the size has a big effect on the throughput. Up to 5 MB the speed is rapidly increasing, but after this point the increase is declining. So we can say that the most efficient chunk size would be 5 MB.

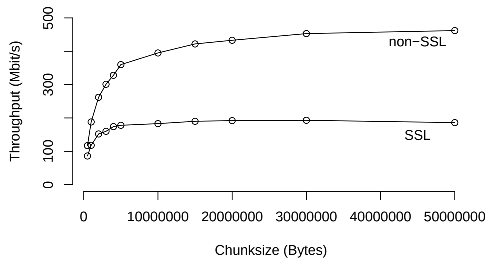

Figure 2: *Chunksize compared to throughput in* ***original*** *implementation.*

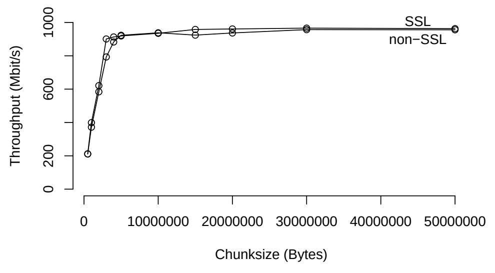

Figure 3: *Chunksize compared to throughput in* ***new*** *implementation.*

In figure 3 and figure 4 we can see the throughput compared to chunksizes for our new upload implementation. The first one shows this comparison without delays, and for the latter one a 100ms delay was introduced.

From these figures we can derive that for our new implementation there is almost no difference between SSL and non-SSL. But, introducing an extra latency has a negative effect on performance.

5.4.2. SSL Cipher

As we have shown in section 4.3, choosing the right cipher for your SSL connection is essential when you want to utilize all available bandwidth. A lot of re-

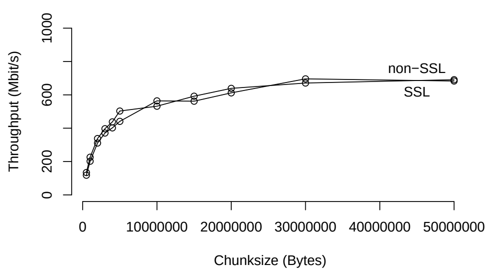

Figure 4: *Chunksize compared to throughput in **new** implementation with **100ms delay**.*

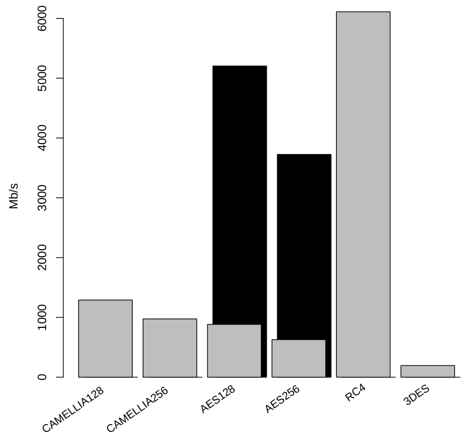

Figure 5: *Comparing different SSL ciphers. The black bars are with AES-NI enabled.*

search has been done about SSL, for example Gupta et al. [18] already showed that Elliptic Curve Cryptography increases performance for SSL. Zhao et al. [31] analyzed the anatomy and performance of SSL Processing. In this section we will compare different SSL ciphers and decide which can be used best and which should definitely not be used.

Figure 5 shows the performance of the SSL ciphers we have tested using the `openssl speed` command. The test suite included in OpenSSL.

We ran our first test on OpenSSL compiled *without* AES-NI support. This are the gray bars in the figure. The clear winner is RC4. Fluhrer et al. [16] present several weaknesses in the key scheduling algorithm of RC4 and show that RC4 is completely insecure in a common mode of operation. The good thing is that those attacks do not apply to RC4 based SSL. The first reason is that the encryption keys are generated by SSL by hashing. This ensures that different sessions have unrelated keys. The second reason is that SSL begins the encryption of a packet using the RC4 state from the end of a previous packet and thus does not re-key after each packet.

The second and third place is for Camellia in 128 bit and 256 bit key mode. Seeing this, one would understand why Google Chrome chooses Camellia as cipher when connection to our server. It is just faster than AES.

But when tests performed with OpenSSL compiled *with* AES-NI support we see a huge performance improvement, as represented by the black bars in the graph. This observation makes AES preferable over Camellia if and only if the client computer supports AES-NI.

Ideally Google Chrome should make this decision based on the clients hardware, but apparently it does not. As a server admin it is not easy to determine which is best for your clients, but since AES-NI was introduced in 2010, more and more processors are equipped with it, and this makes it a good choice to have AES as the main encryption algorithm.

To conclude this part, system administrators should take care of configuring their servers properly. RC4 is shown to be fast and still considered secure. It gives the client the best performance since it is hard to determine if a client has AES-NI support.

6. Terabyte Challenge

Since this was the terabyte challenge, and our goal was to transfer one terabyte of data within 5 hours on a low latency path, we tested our implementation and improvements with an one terabyte file.

For testing with a 1TB file we used a regular harddisk connected via eSATA to one of our laptops. This was needed because the internal harddisk was limited to 250GB and therefore cannot contain a 1TB file. We noticed that using a regular harddisk instead of the SSD has a negative effect on performance, but still we are able to reach pretty good throughput. Although the external eSATA harddisk can handle speeds of over 1Gb/s there is still a decrease in performance. We reckon this has to do with the seek time of the regular disk. Because it has to seek for the beginning of each chunk whenever a worker requests a part of the file the real throughput is slower than 1Gb/s. By increasing the chunks size we can increase the throughput, but we can not get it to 1Gb/s. Sadly we did not have a SSD large enough to hold a 1TB file, but as we have shown in section 5.1 higher speeds would be possible with a SSD.

We first did the terabyte challenge within our experimental environment with a network delay of less than 1 ms. The average speed for this transfer was 702Mb/s and it completed in a stunning 3 hours and 19 minutes. This is way less than the goal of 5 hours. We did also a 10GB file upload from the SSD which holds an average speed of 928Mb/s. If we translate this to a 1TB file we can have an upload time of approximately 2.5 hours.

7. Conclusion

We can conclude that there are indeed bottlenecks in the current FileSender system. The main problem for the low throughput lies in the implementation of the file upload. Waiting on a reply from the server between each chunk introduces gaps and these gaps have a big impact on performance.

We have shown that with a good file upload implementation and carefully choosing the right encryption cipher for SSL it is possible to utilize all the available TCP bandwidth of a 1Gbit low latency connection.

As a last thing in this report we have beaten the FileSender terabyte challenge by uploading 1TB of (random) data in under 5 hours.

7.1. Future Work

File Download For this research we did not look into the file download, which is the second step when transferring a file from one user to another. Currently this is handled by a single PHP script that just copies the complete file to the users browser. We expect that by parallelizing this, some performance improvement can be gained here as well.

Harddisk Bottleneck One possible cause that our current bottleneck has, might be caused by the harddisk seek time. If this is the case, we think that a read ahead buffer is able to read more sequentially from the harddisk and might counter this problem.

TCP optimization Also the TCP problems as discussed in section 4.4 and how

TCP optimization affect the throughput speeds will be considered as future work.

Security A final thing to see as future work is the security of our implementation. The current implementation allows for an easy Denial-of-service attack because it does not extensively validate the data it receives from the client. Since the client can easily change the start byte or the chunk size it would be possible to fill up the queue rapidly and thus make the server unusable.

Acknowledgements

We would like to express our gratitude to all those who gave us the possibility to complete this project and report.

Thanks go out to SURFnet, UNINETT and AARNet for offering this great and challenging subject and providing us with all required hardware.

We are grateful to Jan Meijer from UNINETT in Norway for guiding us through this project, thinking with us and challenging us with interesting and thought-provoking questions.

Finally, we also want to thank Xander Jansen of SURFnet and Guido Aben of AARNet who supported us at the SURFnet office during the course of this project.

References

[1] ab - apache http server benchmarking tool. `http://httpd.apache.org/docs/2.2/programs/ab.html`.

[2] Apache http server. `http://httpd.apache.org/`.

[3] Elemeary os. `http://elementaryos.org/`.

[4] Filesender project. `http://www.filesender.org/`.

[5] Fourthxtended filesystem. `https://ext4.wiki.kernel.org`.

[6] Gnu wget 1.13.4 built on linux-gnu. `http://www.gnu.org/software/wget/`.

[7] Google chrome version 24.0.1312.52. `http://chrome.google.nl/`.

[8] Mozilla firefox web browser. `http://www.mozilla.org/en-US/firefox/`.

[9] Nginx, an open source web server and reverse proxy. `http://wiki.nginx.org/Main`.

[10] Sne master research projects 2012-2013. `http://staff.science.uva.nl/~delaat/rp/2012-2013`.

[11] Wireshark: Network protocol analyzer. `http://www.wireshark.org/`.

[12] W. Allcock. GridFTP: Protocol Extensions to FTP for the Grid. 2003.

[13] E. Altman, D. Barman, B. Tuffin, and M. Vojnovic. Parallel TCP Sockets: Simple Model, Throughput and Validation. In *IEEE INFOCOM*, pages 1–12, 2006.

[14] K. Aoki, T. Ichikawa, M. Kanda, M. Matsui, S. Moriai, J. Nakajima, and T. Tokita. Camellia: A 128-bit block cipher suitable for multiple platforms — design andanalysis. In D. Stinson and S. Tavares, editors, *Selected Areas in Cryptography*, volume 2012 of *Lecture Notes in Computer Science*, pages 39–56. Springer Berlin Heidelberg, 2001.

[15] J. Daemen and V. Rijmen. AES Proposal: Rijndael. 1998.

[16] S. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the key scheduling algorithm of rc4. In S. Vaudenay and A. Youssef, editors, *Selected Areas in Cryptography*, volume 2259 of *Lecture Notes in Computer Science*, pages 1–24. Springer Berlin Heidelberg, 2001.

[17] S. Gueron. *Intel's New AES Instructions for Enhanced Performance and Security*. 2009.

[18] V. Gupta, S. Gupta, S. Chang, and D. Stebila. Performance analysis of elliptic curve cryptography for ssl. In *Proceedings of the 1st ACM workshop on Wireless security*, WiSE '02, pages 87–94, New York, NY, USA, 2002. ACM.

[19] T. J. Hacker, B. D. Athey, and B. Noble. The End-to-End Performance Effects of Parallel TCP Sockets on a Lossy Wide-Area Network. In *International Parallel and Distributed Processing Symposium/International Parallel Processing Symposium*, 2002.

[20] T. J. Hacker, B. D. Noble, and B. D. Athey. The effects of systemic packet loss on aggregate TCP flows. In *Supercomputing Conference*, pages 1–15, 2002.

[21] T. J. Hacker, B. D. Noble, and B. D. Athey. Improving Throughput and Maintaining Fairness using Parallel TCP. In *IEEE INFOCOM*, volume 4, 2004.

[22] S. Hemminger. Network emulation with netem. In *LCA 2005, Australia's 6th national Linux conference (linux.conf.au)*, Sydney NSW, Australia, Apr. 2005.

[23] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323 (Proposed Standard), May 1992.

[24] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. *SIGCOMM Comput. Commun. Rev.*, 32(4):89–102, Aug. 2002.

[25] R. Klomp and E. Schaap. Filesender implementation. `https://github.com/OS3/filesender-challenge/commit/7189be8a`.

[26] R. Klomp and E. Schaap. Library: Bambus uploader. `https://github.com/leaf26/bambus`.

[27] T. V. Lakshman and U. Madhow. The performance of TCP/IP for networks with high bandwidth-delay products and random loss. *IEEE/ACM Transactions on Networking*, 5:336–350, 1997.

[28] S. Lim, G. Fox, A. Kaplan, S. Pallickara, and M. Pierce. Gridftp and parallel tcp support in naradabrokering. In M. Hobbs, A. Goscinski, and W. Zhou, editors, *Distributed and Parallel Computing*, volume 3719 of *Lecture Notes in Computer Science*, pages 93–102. Springer Berlin Heidelberg, 2005.

[29] D. Lu, Y. Qiao, P. A. Dinda, and F. E. Bustamante. Modeling and Taming Parallel TCP on the Wide Area Network. In *International Parallel and Distributed Processing Symposium/International Parallel Processing Symposium*, 2005.

[30] NLANR/DAS. Iperf, a modern alternative for measuring maximum tcp and udp bandwidth performance. `http://iperf.sourceforge.net/`.

[31] L. Zhao, R. Iyer, S. Makineni, and L. Bhuyan. Anatomy and performance of ssl processing. *Performance Analysis of Systems and Software, IEEE International Symmposium on*, 0:197–206, 2005.

A. Benchmark Results

A.1. Hard Disk Benchmark

Device	Operation	Test 1	Test 2	Test 3	Test 4	Test 5	Average	SD
Server	Read	405	396	373	403	411	397.60	14.76
	Write	503	506	550	525	490	514.80	23.32
Laptop with HDD	Read	101	103	104	105	103	103.20	1.48
Laptop with SSD	Read	464	468	475	474	458	467.80	7.09

Table 2: *Disk read and write speeds are presented in MB/s. The dd command is used for these test. Read: dd if=randomdata.bin of=/dev/null and write: dd count=1M bs=60k if=/dev/zero of=/tmp/test.img. The randomdata.bin file is a 10GB file of random data.*

A.2. Network Throughput Benchmark

From	To	Test 1	Test 2	Test 3	Test 4	Test 5	Average	SD
Server	Client	944	941	942	941	940	941.60	1.52
Client	Server	938	934	935	934	936	935.40	1.67

Table 3: *Network throughput in MBit/s. The recieving machine is setup with iperf -s and the sending machine is setup with iperf -c 192.168.0.10 -i 10 -t 50*

A.3. SSL Benchmark

Cipher	Test 1	Test 2	Test 3	Test 4	Test 5	Average	SD
CAMELLIA128	1.293	1.289	1.286	1.290	1.292	1.289,98	3,01
CAMELLIA256	975	972	975	975	974	974,36	1,27
AES128	882	882	884	883	883	883,06	0,89
AES128 with AES-NI	5.202	5.207	5.198	5.209	5.201	5.203,51	4,24
AES256	628	630	629	626	628	628,26	1,39
AES256 with AES-NI	3.727,20	3.728,00	3.719,58	3.727,04	3.724,31	3.725,23	3,45
RC4	6.117	6.111	6.111	6.104	6.113	6.111,02	4,63
3DES	194	195	194	194	194	194,35	0,18

Table 4: *SSL benchmark in Mbit/s. Command used: openssl speed -evp <cipher>. Results from 1024 block size are compared.*

B. Chunk Sizes

B.1. Default Installion

Size (in Bytes)	With SSL (Mb/s)	Without SSL (Mb/s)
500.000	86	117
1.000.000	118	188
2.000.000	152	262
3.000.000	160	301
4.000.000	174	328
5.000.000	178	360
10.000.000	183	395
15.000.000	190	422
20.000.000	192	433
30.000.000	193	453
50.000.000	186	462

Table 5: *Chunksize compared to throughput on default installation of FileSender*

B.2. Improved Implementation

Size (in Bytes)	With SSL (Mb/s)	Without SSL (Mb/s)
500.000	212	211
1.000.000	399	372
2.000.000	621	584
3.000.000	901	793
4.000.000	913	882
5.000.000	919	923
10.000.000	935	938
15.000.000	958	924
20.000.000	961	937
30.000.000	966	957
50.000.000	963	956

Table 6: *Chunksize compared to throughput on improved implementation of FileSender using 6 workers on a low latency path (<1ms)*

Size (in Bytes)	With SSL (Mb/s)	Without SSL (Mb/s)
500.000	118	134
1.000.000	202	226
2.000.000	311	338
3.000.000	371	397
4.000.000	402	438
5.000.000	441	503
10.000.000	564	532
15.000.000	562	592
20.000.000	612	639
30.000.000	696	671
50.000.000	683	691

Table 7: *Chunksize compared to throughput on improved implementation of FileSender using 6 workers on a higher latency path (100ms RTT)*

C. Materials

C.1. Hardware

C.1.1. Server

Brand	Dell
Model	PowerEdge R420
CPU	INTEL XEON E5-2430L (2x)
Memory	8GB RDIMM, 1333 MHZ (4x)
Disk	500GB, NEAR-LINE SAS 6GBPS, 2.5-IN, 7.2K (6x)
Interface	Integrated 10/100/1000 Mbps NIC (2x)
OS	Ubuntu Server 12.10

C.1.2. Client 1

Brand	Dell
Model	Latitude E6220
CPU	INTEL CORE I7-2640M
Memory	8GB (2X4GB) 1333MHZ DDR3
Disk	256GB MOBILITY SOLID STATE
Display	12.5 inch ULTRASHARP HD
Interface	Integrated 10/100/1000 Mbps NIC
OS	ElementryOS

C.1.3. Client 2

Model	Latitude E6220
CPU	INTEL CORE I7-2620M
Memory	8GB (2X4GB) 1333MHZ DDR3
Disk	250GB SERIAL ATA (7,200 RPM)
Display	12.5 inch ULTRASHARP HD
Interface	Integrated 10/100/1000 Mbps NIC
OS	ElementryOS

D. Simplified Algorithms

This section of the appendix contains a simplified pseudo code of our solution. The full source code can be found online in the repository of the project??.

D.1. Client Side

Algorithm 1: Client Side: Main

```
input: numberOfWorkers, chunkSize, sourceFile
currentStartByte ← 0;
activeWorkers ← 0;
for 1 to numberOfWorkers do
    // This will only create the worker.
    // Worker is started later
    createWorker();
    activeWorkers ← activeWorkers + 1;
end
while true do
    // When a worker notifies that it is ready, do some actions for
       that particaular worker.
    if worker ready then
        activeWorkers ← activeWorkers - 1;
        if currentStartByte < filesize(sourceFile) then
            if currentStartByte + chunkSize > filesize(sourceFile) then
                endByte ← filesize(sourceFile);
            else
                endByte ← currentStartByte + chunkSize;
            end
            worker.uploadChunk(currentStartByte, endByte, sourceFile);
            currentStartByte ← currentStartByte + chunkSize;
            activeWorkers ← activeWorkers + 1;
        else
            if activeWorkers == 0 then
                // All workers are done.  Finish file by sending an
                   empty chunk.
                sendEmptyChunk();
                return;
            end
        end
    end
end
```

Algorithm 2: Client Side: Worker

```
  input: startByte, endByte, sourceFile
1 blob ← sourceFile.slice(startByte, endByte);
  // Set header and upload blob
2 setHeader('X-Start-Byte',startByte);
3 uploadChunk(blob);
4 while not received reply do
5 |   wait
6 end
7 process reply;
8 notify main;
```

D.2. Server Side

Algorithm 3: Server side out-of-order handling

```
input: destinationFile, chunkData, X-Start-Byte
lock (destinationFile);
filesize ← filesize(destinationFile);
if X-Start-Byte equals filesize then
    destinationFile.append(chunkData);
else
    // We cannot append our chunk, so try to append chunks from the
       queue first
    foreach chunk in queue as queue-item do
        if filesize equals queue-item['startByte'] then
            destinationFile.append(queue-item);
            remove queue-item from queue;
        else
            if filesize > queue-item['startByte'] then
                // This should not happen...
                // Remove item from the queue
                remove queue-item from queue;
            else
                exit foreach loop: we can never append the rest of the chunks;
            end
        end
    end
    // Check again whether we can append our chunk
    if X-Start-Byte equals filesize then
        destinationFile.append(chunkData);
    else
        store chunk in queue;
    end
end
unlock (destinationFile)
```