Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Revisiting bulk data transport over HTTP

F. Hernandez[a]

IN2P3/CNRS computing center, Lyon, France

Abstract

In this white paper we present preliminary results of our work which aims at evaluating the suitability of HTTP-based software tools for transporting scientific data over high latency network links. We present a motivating use case, the tools used for this test and provide some quantitative results and the perspectives of this work.

Introduction

Moving massive amounts of data among geographically distant sites is a typical requirement of nowadays scientific experiments. Long haul networks linking instruments and data processing sites are commonly used in the distributed platforms designed for processing the data handled by modern scientific experiments.

Experimental data is typically collected at the site where the instrument is located and then transported to one or more data processing sites for archival and processing purposes. Specific tools have been developed over the years for efficient transport of data over high latency network links [1], [2], [3], [4], [5]. Some of those tools extend a standard protocol (e.g. FTP [6]) and other tools are custom-built for the purpose of transporting data over network links in an efficient way.

The standard Hypertext Transfer Protocol, HTTP [7], extensively used by web applications, has been traditionally avoided for bulk transfer of scientific data. In this work, we report on our experience evaluating a modern implementation of HTTP as the protocol for transporting data over long distance network links.

In the rest of the document we describe our use case of interest, we present how we use HTTP in this evaluation, the testbed platform and the software used for this work and finally present our preliminary results and perspectives.

Motivation

During 10 years starting in 2022, a multi-terabyte data set of raw images of the southern sky taken every night by the imaging device of the Large Synoptic Survey Telescope (LSST) [8] will be transported from the acquisition site in the Chilean mountains to the main data processing and archival site located at the National Center of Supercomputing Applications (NCSA) [9] in the USA. One copy of the full dataset will be transferred from there to IN2P3 computing center [10], the satellite site in France, for permanent storage and on-site processing. The images and astronomical catalog resulting from the annual processing jointly performed at both sites will be exchanged for cross verification and permanent archival. The processed data will eventually be made available to

[a] Contact the author by email: fabio@in2p3.fr

the research community of the institutions participating in the project and to the astronomy research community at large.

Raw data produced by the telescope are expected to be stored in the form of files of sizes in the range of 100 MB to a few gigabytes. The format of the files may vary but for the purpose of this work a file is considered an opaque string of bytes. Transatlantic network links will be used for exchanging data between the American and French computing centers.

This is an example of a use-case often found in nowadays scientific experiments: a scientific instrument continuously producing significant amounts of data which need to be transported to one or more data processing sites (e.g. high performance or high throughput computing facilities) where data is transformed, analyzed, archived and made available to the community.

The HTTP protocol has several properties which are relevant for this use case and motivate this work. First, it is a well understood standard protocol for which there are several mature implementations in many programming languages. Second, a lot of effort has been devoted by the open source community and by the Internet industry for building and continuously improving an ecosystem of tools developed around this protocol. Given the strategic importance of HTTP for the global web platform as we know it today, it is likely that the protocol itself and the tools built to support its usage will still be relevant for the next decade. And even if not, another protocol and associated tools designed to solve the same problems that HTTP solves would likely be developed and available for us to use.

In addition to those considerations, the available versions of HTTP, namely HTTP/1.1 and the recently standardized HTTP/2 [11], include several intrinsic features that make it an attractive candidate as a transport protocol for bulk file transfer over network links. Secure HTTP, that is, HTTP on top of the Transport Layer Security (TLS) protocol [12], provides built-in confidentiality of the communication channel and data integrity checking. It also supports several authentication mechanisms, including but not limited to standard X.509 certificates.

Evaluation tools

The architecture of HTTP consists of a client and a stateless server. The HTTP server understands a limited number of actions, referred to as *verbs* (e.g. GET, POST, HEAD, …), that the client may request the server to perform on a well identified resource. The resources the server acts upon are previously agreed and known to both client and server. Examples of the actions that the client may request the server to perform are to send (download) or to receive (upload) an opaque sequence of bytes.

For the purposes of this evaluation, we deliberately restrict our usage of HTTP to a very simple case: the client asks the file server to respond with the sequence of bytes composing a file, given an arbitrary file identifier and the size of the file it expects the server to send. To fulfill the client’s request, the server generates the requested number of bytes and sends them to the client in the body of the HTTP response. Therefore, no disk input/output operations are performed: since we are interested in measuring the ability of the protocol to efficiently transport data we limit our operations to memory-to-memory data transfers.

Although very simple, this model of operations represents well a general HTTP-based data transfer platform. In such a platform, a file server exposes the data available to its authenticated or anonymous clients to download. When a client needs to download some data, it emits a download request to the server, receives and save the data in permanent storage.

This simple pull model of operations is in contrast to the current practice for scientific bulk data transfer, where a push model is often used. In a push model, the entity wanting to transfer data to a remote location typically sends (i.e. pushes) the data to a remote endpoint. Although the push model is implementable also on top of HTTP, in this evaluation we focus on the pull model.

As part of the download operation, at the client’s request, the server may compute a checksum of the file contents and include it in the response it sends to the client. The client may compute the checksum of the data it receives and compare it to the checksum sent by the server.

The TLS protocol, used by secure HTTP, guarantees the integrity of the data transferred. The checksum computation feature is however useful if the client wants to permanently store the checksum alongside with the file contents, which is a good practice in scientific experiments dealing with large amounts of data files. Computing the checksum on the fly when the file is transferred avoids reading the data back from permanent storage for this specific purpose later on.

We implemented both a client and a server in the Go programming language [13]. Our implementation is available online [14]. Our client and file server both use the secure HTTP implementation built in the Go standard library. The Go standard library includes client implementations of both HTTP/1.1 and HTTP/2. In this work, we report on our usage of both versions.

Our choice of Go as the programming language for this evaluation work is motivated by several reasons. As stated above, Go includes in its standard library an implementation of both HTTP client and servers, so no additional dependencies are needed. The Go compiler generates ready-to-install, standalone executables for multiple hardware platforms and operating systems. Furthermore, Go exhibits a programming model that makes it easier to implement concurrent applications, which we intensively use in our tools. Having the possibility to program both client and server allows for implementing specific features useful for our use case of interest. For instance, as opposed to use an integrated HTTP server, we can easily program the server or the client to use several kinds of storage platforms (such as an object store or a tape library) as the storage source or destination of the data or perform some pre- or post-processing operations (e.g. compaction) when the data is sent by the server or received by the client.

The software tools that we implemented for this evaluation allow for generating a synthetic load on a file server which understands HTTPS (secure HTTP). The client emits a configurable number of simultaneous download requests over a configurable period of time, receives the data sent by the server and summarizes the observed results, in terms of volume of data downloaded, number of requests emitted, download throughput per network connection and aggregated over all the connections as well as download errors. The mean size of the files can also be specified: the client will then generate requests using a file size drawn from a Gaussian distribution with the specified average and a standard deviation of 20% of the specified average file size.

This synthetic load represents well our needs of a hypothetical HTTP-based file transfer platform. In such a platform, where several hundreds of files may be simultaneously downloaded at any given time, we are interested in optimizing the aggregated download throughput, as opposed to optimizing the latency for downloading an individual file.

Testing environment

We performed our tests in two complementary test environments: a local area network (LAN) and a wide area network (WAN). The LAN testbed is composed of 4 well configured computers each with a single 10 Gbps network card (see details of the hardware configuration in the annex). The 4 machines were all connected to the same network switch, with an average latency of 97 µsecs among any pair of them.

For the tests over long distance network, we used a non-dedicated transatlantic network link connecting CC-IN2P3 in Lyon, France to NCSA in Urbana-Champaign, USA. The bandwidth of this link is limited to 10 Gbps and its round-trip time is 110 ms. The connectivity between the two sites was provided by RENATER [15], GÉANT [16] and Internet2 [17].

All the computers involved in our tests run the CentOS Linux distribution. The computers at CC-IN2P3 run CentOS 7 and the single virtual machine at NCSA runs CentOS 6.

We performed two kind of tests. In the CC-IN2P3 local area network, we ran a campaign of tests to confirm that there is no noticeable limit on our ability to use the available bandwidth between the machines of our testbed with our evaluation software suite. We demonstrated that our implementation of the client and server was able to use the available bandwidth of each host in the controlled environment of the LAN, that is, 10 Gbps.

Performing network throughput tests on an uncontrolled, shared, multi-provider, long distance, production network environment is known to be a rather difficult task. Since we don't have the possibility to monitor in real-time the end-to-end usage of the link, we don't have absolute transfer throughput figures to compare against. Therefore, we need to compare the throughput observed by our HTTP-based software tools to the throughput obtained by mature tools, well known for their ability to exploit the available bandwidth of high latency network links, when used in comparable conditions. Specifically, for this work, we used iperf3 [18] and bbcp. iperf3 is developed and maintained by ESNet and is extensively used by Perfsonar [19], the software suite promoted by GÉANT for measuring network performance and routinely used for collecting network health data by the sites participating in CERN's Large Hadron Collider computing grid [20]. bbcp was developed and is maintained by US DOE's SLAC National Accelerator Laboratory and is one of the recommended tools to transfer data over long distance network links.

Let us restate that our goal in this work is not to compare the features of several data transfer software tools, but rather to understand whether a HTTP-based transfer software tool is a competitive alternative to those well-

established tools, for the specific use case we are interested in. For instance, all our data transport tests are performed using secure HTTP (i.e. HTTP over TLS), which guarantees that data is transported over a secure channel which in addition guarantees data integrity. This is not the case of iperf3, so our comparison may strictly speaking not be fair, but intends to provide guidance before deciding to invest more effort developing a software tool.

We present, in the next section, preliminary quantified results of the tests we have performed so far.

Quantified results

Fig. 1 presents a comparison of using the memory-to-memory data transfer throughput observed using netperf [21], a deliberately simple data transfer tool developed in Go language, between two hosts located in the local area network. Each of the two hosts has a single 10 Gbps network card.

This figure shows that, when network latency is low as is the case of hosts connected to a local area network, there is no noticeable penalty associated to using TLS on top of TCP as opposed to using TCP only. In our experience, this is true only when using modern CPU hardware that supports a hardware-assisted implementation of AES, the algorithm used by TLS for encrypting the communications channel. This is not a limitation in practice, since most Intel CPUs sold since about year 2010 and used by high-end servers include hardware instructions used for implementing AES.

The second conclusion that we can draw from this graphic is that a network application developed in Go can use the available network bandwidth when transferring data in the LAN. The CPU usage of the Go application was higher than the one used by iperf3, though.

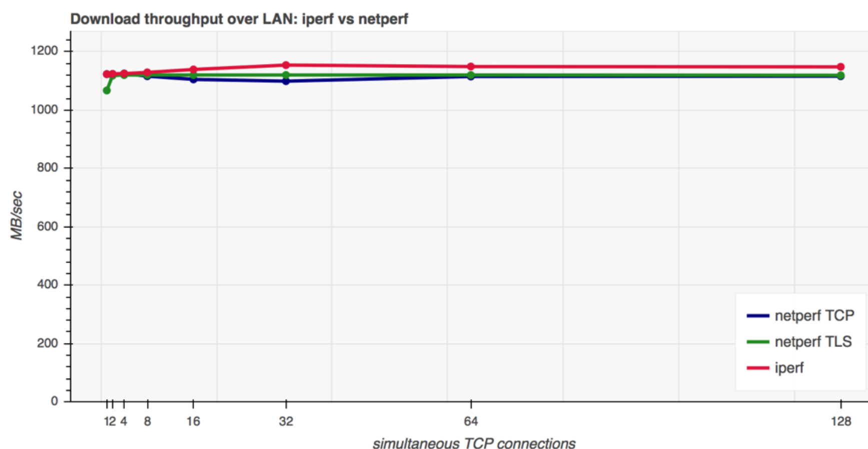

Fig. 1 Comparison of memory-to-memory data transfer throughput between two hosts, each using a 10 Gbps NIC, both located in a local area network (RTT 97 µsec), when using iperf3 and netperf (both TCP and TLS). The throughput is aggregated over all the simultaneous TCP connections.

Fig. 2 shows a comparison of aggregated memory-to-memory throughput observed by using iperf3, bbcp and secure HTTP/1.1 (i.e. HTTP/1.1 over TLS) between two machines connected by a bottleneck link of 10 Gbps, with a RTT 110 msec. As the number of simultaneous TCP connections increases, the observed aggregated throughput of secure HTTP increases beyond the one reached by specialized tools such as iperf and bbcp. The maximum number of TCP connections supported by iperf3 is 128 and by bbcp is 64.

To obtain these figures, 5 tests were performed with each tool for each value of simultaneous TCP connections and the average of the aggregated throughput (i.e. the sum of throughput over all the connections) was computed and shown in the graphic. The two same hosts were used for all these tests and the same network parameters (e.g. TCP window size, TCP congestion algorithm, MTU, etc.) were also kept unmodified. Sizing the TCP window size was deliberately left to the TCP implementation of the Linux kernel, from a sensible initial size set at the system level. In addition, as all the other tests we performed for this work, these tests were all performed over a production, shared transatlantic network link.

It is worth noting that bbcp does not inherently allow for memory-to-memory transfers. For performing this test with bbcp we used a file stored in a RAM disk at the sending site and /dev/null at the receiving site. The comparison

is therefore not strictly fair, but the overhead of using a RAM-based file at the emitting site does not fully explain by itself the observed difference in throughput.

Fig. 3shows a comparison of throughput observed when transferring data using secure HTTP/1.1 and HTTP/2, both using the built-in Go language implementations. HTTP/2 was recently standardized (May 2015) and includes features such as request multiplexing and binary framing. The throughput observed when using HTTP/2 is significantly lower than the one using the more mature HTTP/1.1 implementation. The cause of this difference is not totally understood and requires a thorough investigation.

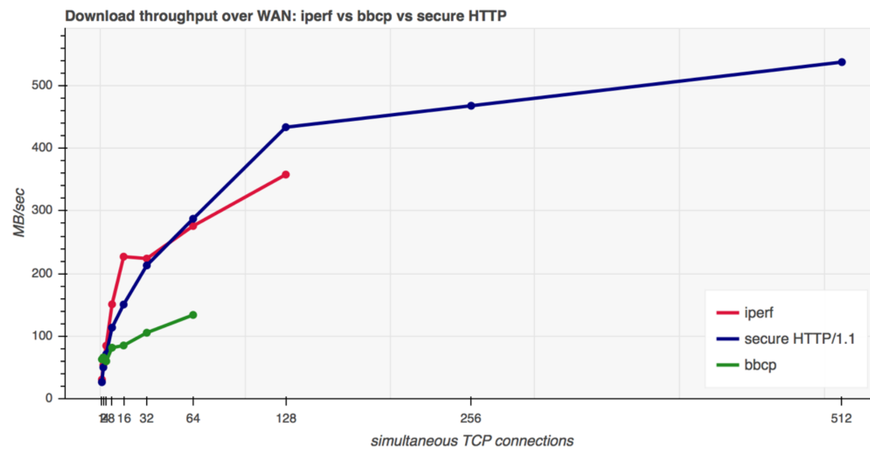

Fig. 2 Comparison of aggregated, memory-to-memory, download throughput by iperf, bbcp and secure HTTP (i.e. HTTP over TLS) between two hosts connected by a bottleneck link of 10 Gbps and a RTT of 110 msec.

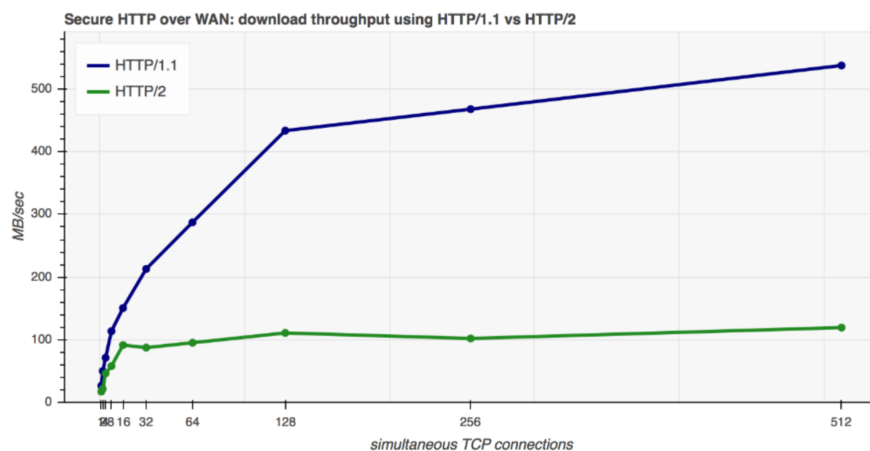

Fig. 3 Comparison of aggregated, memory-to-memory, download throughput observed using secure HTTP/1.1 and secure HTTP/2 between two hosts, connected by a bottleneck link of 10 Gbps and a RTT of 110 msec.

Conclusion and Perspectives

Preliminary results presented above support the idea that modern implementations of HTTP may be a viable alternative to specific tools for bulk transport of files over high latency network links. Programmability of both HTTP-based client and server as well as other considerations exposed above make HTTP an attractive proposition to consider when designing a generic data transfer service from large-scale scientific instruments to data processing centers. Using a ubiquitous, secure, standard data transport protocol removes the dependencies and deployment complexities of custom-built or non-standard tools.

However, we want to stress the fact that although these results are encouraging they are still preliminary. Further testing and comparison with other well established tools for data exchange such as gridFTP and FDT is considered required to take an informed decision.

In the future, we intend to perform additional tests under other network conditions, in particular, when the latency between the two hosts exchanging data is a few tens of milliseconds, as is the case for two sites located in the same country or within nearby countries in Europe.

Performing tests during longer periods is also desirable to improve the statistic, but in practice this is difficult to perform with a non-dedicated network link. We therefore plan to perform additional tests in the local area network by introducing artificial latency and study the reproducibility of our results.

Annex

The relevant hardware components of each machine in our testbed are as follows:

- CPU: 2 x Intel Xeon E5-2623 v4 2.6GHz, 10M Cache, 8.00GT/s
- RAM: 64 GB
- Network: Intel X520 10 Gb DA/SFP+

References

[1] GridFTP, http://toolkit.globus.org/toolkit/docs/latest-stable/gridftp
[2] bbcp, http://www.slac.stanford.edu/~abh/bbcp
[3] bbftp, http://doc.in2p3.fr/bbftp
[4] Fast Data Transfer – FDT, http://monalisa.caltech.edu/FDT
[5] IBM Aspera File Sharing Suite, http://www-03.ibm.com/software/products/en/file-sharing-suite
[6] File Transfer Protocol FTP, https://tools.ietf.org/html/rfc959
[7] Hypertext Transfer Protocol – HTTP/1.1, https://tools.ietf.org/html/rfc2616
[8] Large Synoptic Survey Telescope LSST, http://lsst.org
[9] National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, http://www.ncsa.illinois.edu
[10] IN2P3/CNRS computing center CC-IN2P3, http://cc.in2p3.fr
[11] Hypertext Transfer Protocol Version 2 (HTTP/2), http://httpwg.org/specs/rfc7540.html
[12] Transport Layer Security (TLS) Protocol, version 1.2, https://tools.ietf.org/html/rfc5246
[13] The Go programming language, https://golang.org
[14] chasqui, https://github.com/airnandez/chasqui
[15] RENATER, http://renater.fr
[16] GÉANT, http://www.geant.org
[17] Internet2, http://www.internet2.edu
[18] iperf3, http://software.es.net/iperf/
[19] Perfsonar, http://www.perfsonar.net
[20] LHC computing grid, http://wlcg.web.cern.ch
[21] netperf, https://github.com/airnandez/netperf

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EU's Horizon 2020 research and innovation programme (2014-2020) under grant agreement 653838.

Special thanks to RENATER, GÉANT and Internet2 for providing the end-to-end international connectivity used for this work and to NCSA for allowing us to use its infrastructure for performing these tests.