Investigating the Overhead of the REST Protocol when Using Cloud Services for HPC Storage

Frank Gadban[1], Julian Kunkel[2], and Thomas Ludwig[3]

[1] University of Hamburg, 20146 Hamburg, Germany
frank.gadban@studium.uni-hamburg.de
[2] Reading University, Reading, UK
[3] DKRZ, 20146 Hamburg

Abstract. With the significant advances in Cloud Computing, it is inevitable to explore the usage of Cloud technology in HPC workflows. While many Cloud vendors offer to move complete HPC workloads into the Cloud, this is limited by the massive demand of computing power alongside storage resources typically required by I/O intensive HPC applications. It is widely believed that HPC hardware and software protocols like MPI yield superior performance and lower resource consumption compared to the HTTP transfer protocol used by RESTful Web Services that are prominent in Cloud execution and Cloud storage. With the advent of enhanced versions of HTTP, it is time to reevaluate the effective usage of cloud-based storage in HPC and their ability to cope with various types of data-intensive workloads. In this paper, we investigate the overhead of the REST protocol via HTTP compared to the HPC-native communication protocol MPI when storing and retrieving objects. Albeit we compare the MPI for a communication use case, we can still evaluate the impact of data communication and, therewith, the efficiency of data transfer for data access patterns. We accomplish this by modeling the impact of data transfer using measurable performance metrics. Hence, our contribution is the creation of a performance model based on hardware counters that provide an analytical representation of data transfer over current and future protocols. We validate this model by comparing the results obtained for REST and MPI on two different cluster systems, one equipped with Infiniband and one with Gigabit Ethernet. The evaluation shows that REST can be a viable, performant, and resource-efficient solution, in particular for accessing large files.

Keywords: HPC · Cloud · Convergence · HTTP2 · RESTful APIs · HTTP3 · Storage.

1 Introduction

High-Performance Computing (HPC) utilizes clusters of powerful and fast interconnected computers that can handle complex and data-intensive computational problems. These systems are managed by batch schedulers [41] where user jobs

are queued to be served based on resource usage and availability and without any visibility or concerns on the costs of running jobs. Due to various factors, Cloud Computing [42] gained popularity over the last decade. This has led to the emergence of the *HPC Cloud*, where Cloud providers offer high-end hardware platforms and software environments to run HPC applications.

Due to its simplicity, reliability, flexibility, and consistency, HTTP is the de facto standard for accessing object storage like Amazon S3 [2], OpenStack Swift, and EMC Atmos. A wide adoption of cloud storage in HPC requires the evaluation of the suitability of using HTTP in the HPC environment as an alternative to HPC-native communication protocols like MPI.

In this work, we first provide a detailed examination of the HTTP protocol and its performance in terms of latency and throughput under different conditions for accessing remote data. Secondly, we elaborate on an analytic performance model for data transfer over several protocols, this model allows us to compare current and future protocols in a common framework and will help us predict protocol performance in different hardware environments. We perform several benchmarks comparing MPI to HTTP and use our model to validate the obtained results. Finally, and based on the evaluation, we pinpoint the cause of the HTTP overhead and find that TCP is not the ideal transport protocol for HTTP and that new versions of the HTTP protocol, like HTTP3 which uses UDP, might accelerate the usage of cloud storage in HPC. The structure of this paper is as follows: Section 2 represents the related work. Section 3 describes the test scenarios and defines the relevant metrics that will be addressed using our benchmarks. Section 4 describes the experimental procedure, the used systems, and the methodology of the evaluation conducted in this work. Section 5 analyzes the obtained results. The last section summarizes our findings.

2 Related Work

In the world of HPC, computational performance has long exceeded the performance of the traditional file-centric storage systems since the POSIX file system interface was hardly suitable for data management on supercomputers[56]. Many workarounds to address this issue were proposed, some of them tried to introduce evolved I/O algorithms in MPI, like Data aggregation/sieving in ROMIO, [51] or to implement different data organizations on the back-end storage, like PLFS [3] or to introduce richer data formats for example HDF5, NetCDF [19]. Eventually and although a file represents a convenient way to store the data, the ideal concept for scientific computing/HPC would be rather the use of a data object model[38] where all levels of metadata are encapsulated. Object storage organizes information into containers of flexible sizes, referred to as objects. Each object includes the data itself as well as its associated metadata and has a globally unique identifier. Objects are often accessed directly from the client application, usually using a RESTful API [49]. As such, any comprehensive performance study of an object storage system should take into consideration the latency introduced by a RESTful system. Many researchers have tried to solve data transfer issues through HTTP; some [35] proposed encapsulating TCP data

in UDP payloads, others [12] proposed a dynamic connection pool implemented by using the HTTP Keep-Alive feature to maximize the usage of open TCP connections and minimize the effect of the TCP slow start. Intel®is marketing DAOS [40] as the ultimate Open Source Object Store, nonetheless with a high vendor Lock-in potential since the promised performance can only be achieved on its own proprietary Optane [55] storage Hardware. Since Infiniband [1] is one of the most commonly used interconnects in HPC, the performance of IP over Infiniband [4,23] has been thoroughly studied, however, the performance of HTTP over IP over IB did not get much attention. Our approach to model and analyze the viability of HTTP over Infiniband using performance counters is explained in the next section.

3 Methodology

The two major efficiency indicators addressed in our study are latency and throughput. To our knowledge, few tried to assess the Performance of a REST Service inside HPC, i.e., within a high-performance network. This is why we introduce a modeling approach, based on performance counters, then we perform an evaluation on a testbed, consisting of a content server and a client application consuming the content. Our benchmark for storage access emulates a best-case scenario (HTTP GET Operation / Read Only Scenario from a "remote" Storage Server) because we only want to test the viability and base performance of REST/HTTP as an enabling technology for an object-store. The model with the performance counters can nevertheless be extended to assess and measure the resource consumption of different object storage implementations. The tools and the accomplished tests will be extensively described in Section 4. To identify the major factors impacting the performance, we vary the underlying hardware and the connection mechanism (Ethernet, Infiniband, RDMA) between the server and the client. Finally, to validate our model, we compare its predictions with the experimentally observed values.

3.1 Performance Model

To define our performance model, many metrics are considered, which depict the used hardware, the software stack, and the network protocol in use. Alongside the standard network metrics, we focus on hardware counters of the CPU namely the number of required CPU cycles to identify the processing cost of a data transfer and the L3 evicted memory, this can be used further to check the memory efficiency of the different implementations. In a first step, we consider TCP as a transport protocol, however, the model is later extended in Section 5 to cover MPI. The metrics involved can be summarized as follows. *Fixed system parameters:*

- R: CPU clock rate in Hz
- rtt: round trip time
- mtu: maximum transfer unit

- mss: maximum segment size, transmission protocol dependent (see eq. (4))
- mem_tp: the memory throughput i.e. speed of data eviction from L3 to main memory.
- eBW [7]: is the effective bandwidth between client and server.

Experiment-specific configurations and results:

- Obj_size: file size transferred from the server to be read by the client.
- Nreq: number of requests achieved in 60 sec
- Ncon: number of open connections kept
- Nthr: number of CPU threads executing the benchmark on the client.

Observable metrics (e.g., using Likwid):

- CUC: number of unhalted cycles on each core
- L3EV:amount of data volume loaded and evicted from/to L3 from the perspective of CPU cores[24,31], L3EVs and L3EVc are for server and client respectively.
- PLR: the packet loss rate, proportional to the number of parallel connections.

In our preliminary model t(request) is the time starting from the sending of the first byte of the request to the time the complete response is received:

$$t(request) = t(client) + t(network) + t(server) \tag{1}$$

where t(client), t(network), t(server) are the time fractions needed by the client, network and server respectively to accomplish the request:

$$t(client) = t(compute) + t(memory) + t(cpu\_client\_busy) \tag{2}$$

$$t(server) = t(compute) + t(memory) + t(cpu\_server\_busy) + t(pending) \tag{3}$$

As a rough estimation of the network throughput when using the TCP protocol, and based on the Mathis et.al. formula [27], while presuming that the TCP window is optimally configured, we can safely assume that:

$$net\_tp = \min\{\frac{mss \cdot C}{RTT \cdot \sqrt{PLR}}, eBW\} \tag{4}$$

where C=1 and mss = mtu-40 in case of Ethernet. From this we can calculate t_net:

$$t\_net = Obj\_size/net\_tp + t\_queuing \tag{5}$$

For the sake of simplicity, we suppose that the routing devices between the nodes don't add any latency and as such we can neglect t_queuing The execution time t(compute) can then be defined as:

$$t(compute) = CUC/R \tag{6}$$

t(memory) is the time to traverse the different memory caches, usually narrowed down to:

$$t(memory) = L3EV/mem\_tp \tag{7}$$

Putting it all together, and in the case of intra-node communication, we can safely assume that:

$$t(request) = \frac{CUCs}{Rs} + \frac{L3EVs}{mem\_tp} + \frac{CUCc}{Rc} + \frac{L3EVc}{mem\_tp} + \frac{Obj\_size}{net\_tp} \quad (8)$$

Generalizing a bit further, we end up with :

$$t(request) = \alpha \cdot rtt + \beta_1 \cdot \frac{CUCs}{Rs} + \beta_2 \cdot \frac{L3EVs}{mem\_tp} + \beta_3 \cdot \frac{CUCc}{Rc} + \beta_4 \cdot \frac{L3EVc}{mem\_tp} + \beta_5 \cdot \frac{Obj\_size}{net\_tp} \quad (9)$$

Where α is a weighting factor ($0 \leq \alpha < 1$) [30], β_i are platform and protocol dependent factors to be evaluated in a later section. As such, many factors can influence the above starting from the application delivering the content which affects server CPU and memory usage, those metrics are also affected by the type of client consuming the data as well as by the networking protocol in use and the path traversed by the data. In the following sections, we validate this model while comparing the performance of HTTP over different types of hardware and connection protocols.

4 Experiments

The tests were performed on two different hardware platforms: the first is the WR Cluster, a small test system, equipped with Intel Xeon 5650 processors and Gbit Ethernet, the second is the Mistral supercomputer [14], the HPC system for earth-system research at the German Climate Computing Center (DKRZ), it provides 3000 compute nodes each equipped with an FDR Infiniband interconnect. The nodes used for the testing are equipped with two Intel Broadwell processors (E5-2680 @2.5 GHz) [33]

4.1 Benchmark and Analysis Tools

The RESTful API is the typical way to realize access to object storage, and as such the tools used in this article were preliminary developed to assess HTTP performance. The first experiment checks the latency introduced by a simple web server serving static files, the setup, shown in fig. 1, consists of the lighttpd web server [34] hosting files of different sizes. These files contain randomly generated data, and are placed initially in the in-memory file system (tmpfs) to minimize any storage-related overhead such as disk drive access time. The tests are accomplished using the wrk2 tool [50]. In the following analysis, we vary

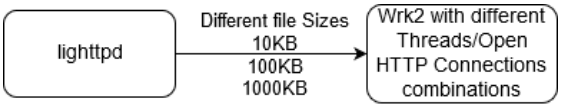


Fig. 1: A simplified overview of the Benchmark Setup

the number of threads and the number of HTTP connections kept open while

trying to keep a steady rate of 2000 requests/second for 60 seconds for each file size.

4.2 Latency

The diagrams in figs. 2a and 2b show the obtained latency distributions.
Observations and interpretations:

- Latency linearly increases with the number of open connections (see Figure 2a), especially true for small file sizes however when the file size grows beyond a certain limit, the number of connections will become irrelevant to the introduced latency (see Figure 2b).
- As shown in fig.2c, for small file size, we observe a latency divergence in particular in the 99 percentile area for bigger file size, we noticed in the case of the 100 KB, the desired request rate of 2000 req/s is not met due to the limitation of the underlying network infrastructure (1Gb/s 125 MB/s)
- It is interesting to note that, in relation to file size (fig.2c), larger files lead to higher memory and network latencies in a way that they can saturate the server's network bandwidth, lowering throughput (see Figure 2b). Therefore, for serving large files, high network bandwidth is more important than compute resources. On the other hand, increasing the number of open Connections (fig.2d) will trigger TCP's congestion mechanism and such they will be competing for the same bandwidth. Increasing the file size as well will cause the Open Connections to lose packets and get stuck waiting for retransmissions.

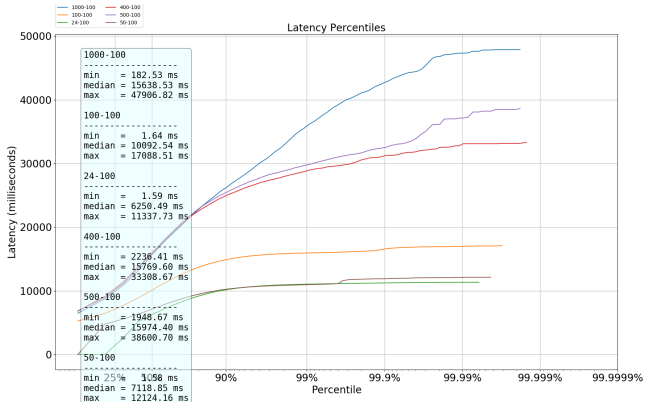
A similar latency distribution is observed when the tests are conducted on the same machine, thus using the optimized[15] loopback interface, where theoretically the network does not pose any throughput bottleneck (iperf [45] result 20 Gbs). From these experiments, we learn that to optimize the throughput, the web requests should not be using different open connections but rather use one or a relatively small number of open Connections and label the web requests accordingly, which is commonly known as HTTP multiplexing [20], where, using the same TCP connection, multiple HTTP requests are divided into frames, assigned a unique ID called stream ID and then sent asynchronously, the server receives the frames and arranges them according to their stream ID and also responds asynchronously; same arrangement process happens at the client side allowing to achieve maximum parallelism.

4.3 Throughput

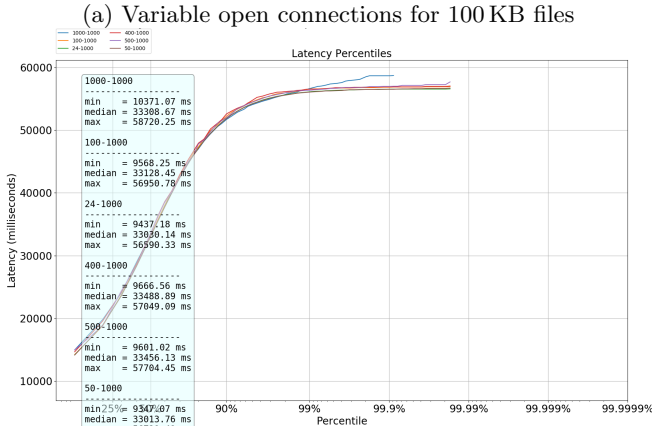
The network throughput of our system is calculated as follow:

$$Throughput = \frac{Nreq \cdot Obj\_size}{time}$$

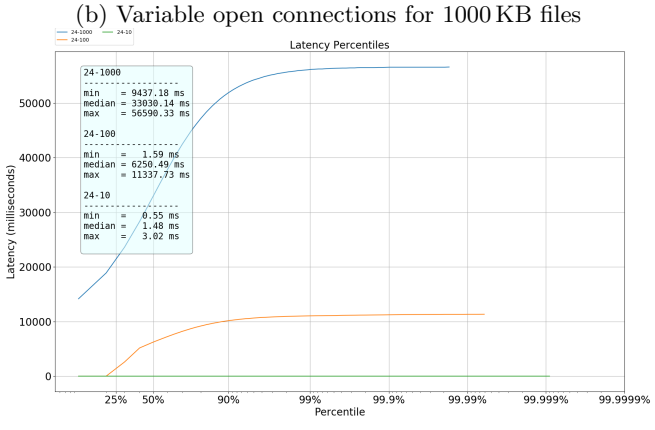
The results are shown in Figure 3. In the case of inter-node communication, an increase in the number of Open Connections will increase the throughput, however, this is only relevant for small file sizes below 1 MB.



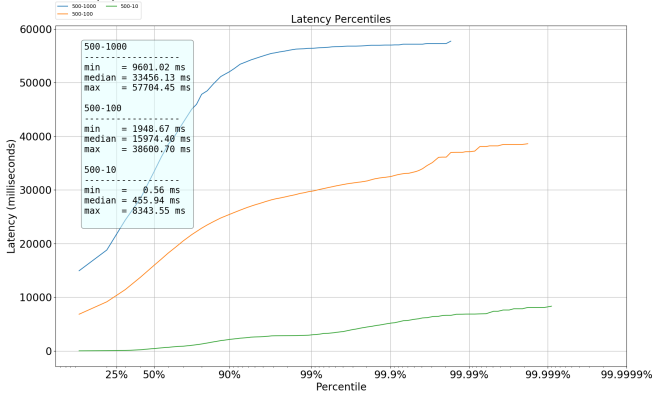
(a) Variable open connections for 100 KB files



(b) Variable open connections for 1000 KB files



(c) Variable file size for 24 open connections



(d) Variable file size for 500 open connections

Fig. 2: Measured latency for different experiments

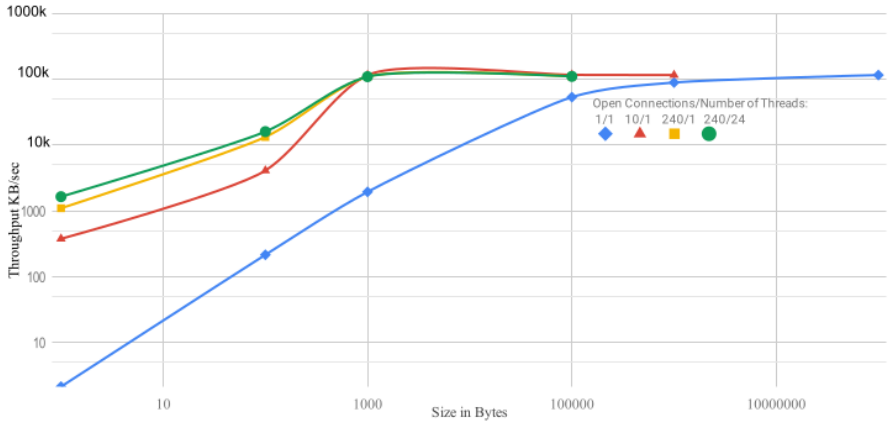
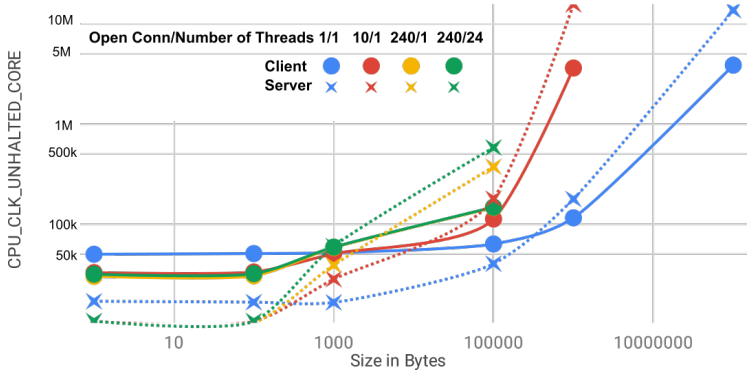


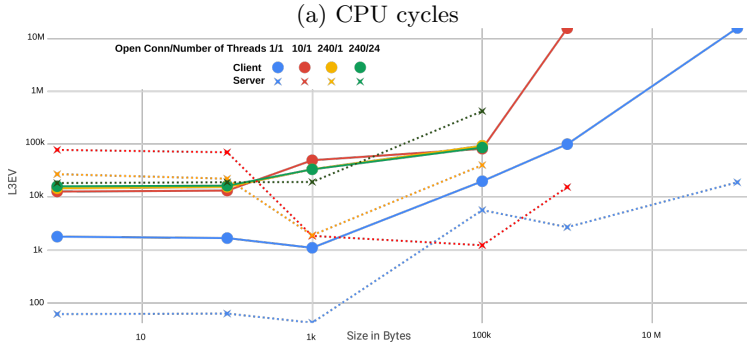
Fig. 3: Throughput in KB/sec for a variable object size and open connections/threads

4.4 Resource Usage Measurements

In addition to the latency diagrams and the findings gained from them, another point to consider is the efficiency of the IO itself, this is why we measure the Memory and CPU usage needed to achieve a certain throughput. To accomplish this, likwid-perfctr [16] is used. It uses the Linux 'msr' module to access the model specific registers stored in /dev/cpu/*/msr (which contain hardware performance counters), and calculates performance metrics, FLOPS, bandwidth, etc, based on the event counts collected over the runtime of the application process. The conducted experiment is similar to Section 4.2, however, this time we are using wrk[21] without specifying a maximum req/s rate, for 1 minute, during this time Likwid is recording the CPU performance counters which are relevant in this scenario. The server application is pinned to one core using Likwid, the same is done on the client side, As wrk tests are performed using only 1 thread, this doesn't impose a performance limitations and ensures that the process is run on the first physical core and not migrated between cores which may lead to overhead. CPU consumption is recorded, CPU_CLK_UNHALTED_CORE is the metric provided by Likwid that represents the number of clock ticks needed by the CPU to do some reasonable work. The instructions required to accomplish one request - by the server as well as by the client - seems to be linear with the file size, as shown in Figure 4a. To note that the server seems to be consuming more CPU cycles as the client to deliver a request, which might be because we are using the lighttpd web server without modifying the default configuration. Note also that over a certain file size limit, the number of timeouts increases since we are approaching the maximum throughput achieved by the system.



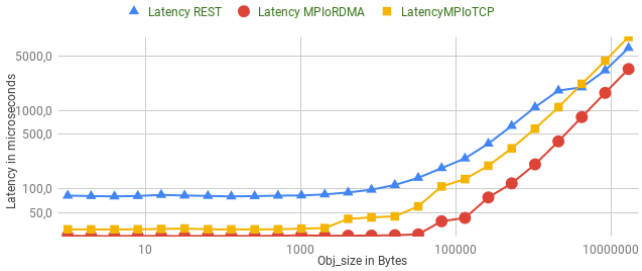
(a) CPU cycles



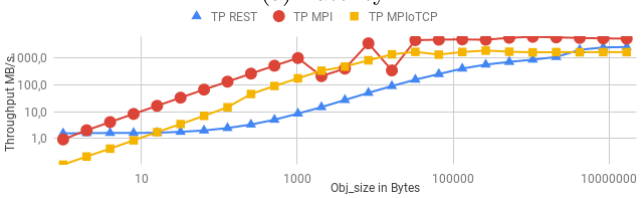
(b) L3 evicted volume

Fig. 4: Likwid metrics for the client and server with a variable size and Open Connections/Threads combination

Regarding memory utilization: Basically, when reading a file (represented by HTTP response), the client needs to store the data received in memory. If the file size exceeds the size of the CPU cache, we expect that data is evicted to main memory, which is measured in L3 cache evictions. This metric is recorded using Likwid and shown in Figure 4b. Basically, we can see that even for 100 MB files only 10 MB of data is evicted on the client. There is no eviction on the server because it sends the data directly to the client. This is an indication that zero-copy [53] is in use on the client and the network interface card offloads the processing of TCP/IP. This allows the network card to store the data directly into the target memory location. Generally, with zero-copy, the application requests the kernel to copy the data directly from a file descriptor to the socket bypassing the copy in user mode buffer and, therefore, reducing the number of context switches between kernel and user mode. Furthermore, when data does not fit in the processor L3 caches (12 MB), the evicted data, i.e., the data passed to memory increases significantly causing a performance drop, curiously the rate of increase of the client evicted memory is greater than on the server, leading us to another interesting conclusion, namely that while most studies focused on optimizing the server-side, it might be the client-side that needs to be addressed.



(a) Latency



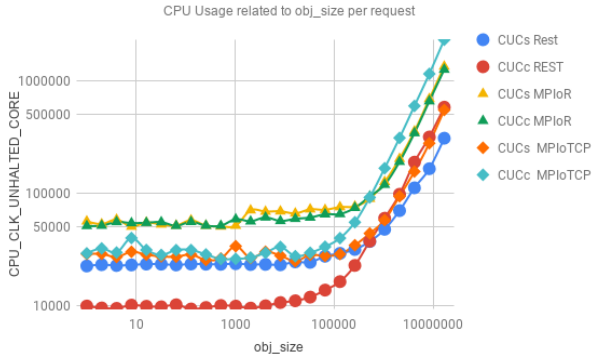
(b) Throughput

Fig. 5: Results for the protocols for a variable file size

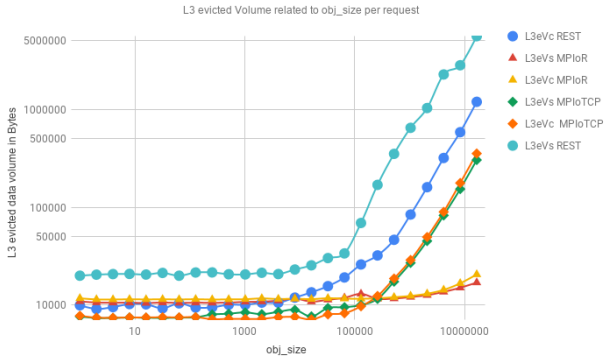
4.5 REST vs. MPI

As found in the previous tests, the available bandwidth plays an important role in determining the latency and the throughput being achieved. The following tests are achieved on Mistral where Infiniband [1] is available. Our next step is to compare the REST protocol with an established data transfer method in the HPC world, namely the Message Passing Interface MPI [52]. To achieve this we launch the same tools used above (likwid+lighttpd) on one node and (likwid+wrk) on another while varying the file size in a power of 2, and recording the different metrics, the transfer takes place over the Infiniband interface. Then we launch the OSU Micro Benchmark [39] alongside with likwid on two nodes using the same file sizes and record the same metrics. The OSU tests are executed over Infiniband, the first time by using RDMA and the second time over TCP. The obtained results are used to plot fig. 5a and fig. 5b. We notice that:

- For small object sizes, the latency of Rest is obviously higher than the one of MPI, as already mentioned in our latency tests, this is due to the HTTP overhead.
- The throughput achieved using MPI is better than the one using REST however when comparing MPI and REST both over TCP, we notice that this is not the case especially for very small and large files which leads to the conclusion that the overhead due to the TCP stack is the main factor slowing down the object storage implementation.
- The performance dip seen in the red line for a file size of above 1 KB is due to the MPI implementation that uses a combination of protocols for the same MPI routine, namely the use of the eager protocol [11] for small messages, and rendezvous protocol for larger messages.



(a) CPU cycles per request



(b) L3 evicted per request

Fig. 6: Likwid metrics for the client and server for the different protocols

- Another particular finding depicted by fig. 6a is that the CPU cycles needed for the sender to push the data when using MPI is higher than by using REST, this becomes visible for file sizes above 100 KB.
- Figure 6b shows that, as expected, the evicted data volume stays constant in the case of MPI over RDMAoIB because of the direct data transfer from server main memory to client main memory. Furthermore, the L3-evicted memory for both REST and MPI over TCPoIB is constant for files smaller than 100 KB but increases exponentially afterward. Presumably, because parts of the protocol such as network packets re/assembly is controlled by the kernel and not the network interface.

4.6 HTTP Overhead

For HTTP 1.1, knowing the amount of bytes_read by the HTTP parser in wrk, and the number of request achieved: $overhead\_per\_request = \frac{bytes\_read}{Nreq} - objsize$

We find that the overhead is about 233 bytes per request, mainly due to the uncompressed, literally redundant, HTTP response headers.

5 Evaluation of the Performance Model

To validate the predictive model defined in eq. (9), we use the values reported by the REST latency Benchmark on Mistral in Section 4.5; the hardware-specific parameters are calculated as follows: Data between sockets and memory is shipped via a 9.6 GT/s QPI interface [33]. According to the Intel QPI specification [32] 16 bit of data are transferred per cycle, thus the uni-directional speed is 19,6 GB/s. The communication protocol has an overhead of roughly 11 The compute nodes of Mistral are integrated in one FDR InfiniBand fabric, the measured bandwidth between two arbitrary compute nodes is 5.9 GByte/s, as such net_tp = 5,9 GByte/s, rtt measured using qperf and found = 0.06ms and mtu = 65520 Bytes. We only need to get the values of the coefficients β_i in eq. (9). This is done by using a regression analysis tool, in this case the one provided by Excel: the obtained R square and F values are examined, for each iteration, to check respectively the fitness and the statistically significant of our model. Finally we calculate the predicted values and we compare them to the ones obtained in the benchmark by determining the error rate using eq. (10). The tables can be found at : https://github.com/http-3/rest-overhead-paper.

$$error\% = (t\_req - t\_req\_calcul) \cdot 100/t\_req \tag{10}$$

In case of RESToTCPoIB, we find that ($\alpha = 1$) , $\beta_1 = \beta_3 = \beta_4 \sim 1$, $\beta_2 = 6$ and $\beta_5 = 3/2$. The deviation (error rate) between the estimated value and the benchmark results is primarily below 10 percent, and indeed in the range of 1 percent for small and large file sizes.Equation (9) yields:

$$t(request) = rtt + \frac{CUCs}{Rs} + 6 \cdot \frac{L3EVs}{mem\_tp} + \frac{CUCc}{Rc} + \frac{L3EVc}{mem\_tp} + \frac{3}{2} \cdot \frac{Obj\_size}{net\_tp} \tag{11}$$

In case of MPIoTCP, we obtain ($\alpha = 0.1$), $\beta_1 = \beta_2 = \beta_3 = \beta_4 \sim 1$, and $\beta_5 = 2.7$. The error rate is less than 5 percent for small and large file sizes.

$$t(request) = 0.1 \cdot rtt + \frac{CUCs}{Rs} + \frac{L3EVs}{mem\_tp} + \frac{CUCc}{Rc} + \frac{L3EVc}{mem\_tp} + 2.7 \cdot \frac{Obj\_size}{net\_tp} \tag{12}$$

In case of MPIoRDMA, we obtain ($\alpha = 0$), $\beta_1 = \beta_3 = 1/2$ and $\beta_2 = \beta_4 = \beta_5 \sim 1$. The error rate is primarily below 10 percent, and less than 5 percent for small and large file sizes.

$$t(request) = \frac{1}{2} \cdot \frac{CUCs}{Rs} + \frac{L3EVs}{mem\_tp} + \frac{1}{2} \cdot \frac{CUCc}{Rc} + \frac{L3EVc}{mem\_tp} + \frac{Obj\_size}{net\_tp} \tag{13}$$

By investigating the model terms, we can infer some general behavior and verify our expectations. The latency for MPIoRDMA is expected to be lower than the others, this is why α is close to 0 for this model. If β_5 is above 1, it is an

indicator that we cannot achieve full network throughput. REST and MPIoTCP show otherwise similar performance characteristics while the MPIoRDMA model is approximated to use half the CUC, which actually means it needed twice as many compared to the TCP models - maybe due to busy waiting. These assumptions can be verified by looking at Figure 6a and Figure 6b. In conclusion, we notice that while TCP proved itself for end-to-end communications over long distances, it is however less suitable for data center networking, mainly because of its processing overhead, hence degrading the aspired performance. On the other side, CPU and Memory consumption for the REST over TCP Model remained adequate in comparison with MPI over TCP and MPI over RDMA.

5.1 Comparing the protocols: HTTP1.1 vs HTTP2 vs HTTP3

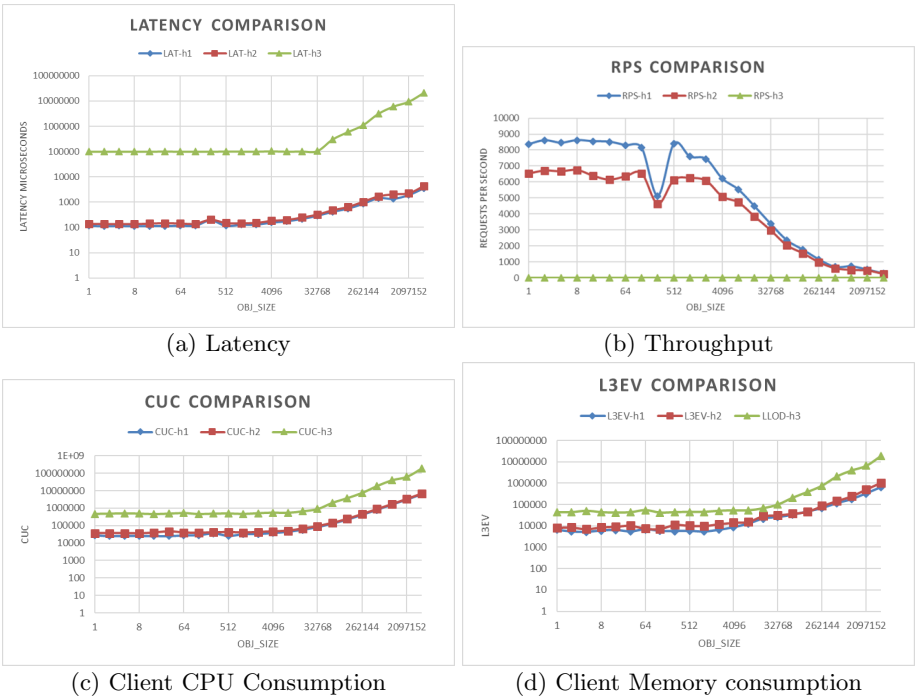
The same setup described in fig. 1 is used here, however the web server, in this case, should be able to deliver the three different protocols. Therefore, openlitespeed [46] is used and the suitable benchmark tool is h2load [26]. To note that we test here the ngtcp2 [44] implementation of HTTP3, because it's TLS library independent, not like other HTTP3 implementations like quiche [10] which requires boringssl. Since at the time of writing, the official OpenSSL Team doesn't support QUIC [47] we use a patched version of OpenSSL provided by the ngtcp2 team. Since HTTP3 didn't achieve the maturity phase yet, we are using the protocols as they are defined in the 27th Draft by the IETF QUIC Working group [29]. The latency and throughput results of the tests on Mistral over InfiniBand are shown in Figure 7a and Figure 7b

Although we are expecting HTTP2 and HTTP3 to perform better than HTTP 1.1, this is not the case. A closer look at the evolution of the parameters defined in our model reveals the cause: Despite the obvious traffic saving of HTTP2, it comes at a considerable memory consumption, which renders the gained advantages negligible. The chosen HTTP3 implementation is circa 10 times more CPU and memory consuming in comparison to the earlier versions, which indicates an implementation issue.

6 Summary

This paper provides a first assessment of using REST as a storage protocol in an HPC environment. A performance model for the relevant HTTP Get/Put operation based on hardware counters is provided and experimentally validated. Our results demonstrate that REST can provide, in many cases, similar latency and throughput to the HPC-specific implementations of MPI while enabling better portability. The developed model covered the general behavior of the different protocols well and was able to generalize and verify the expected behavior.

The new techniques introduced in HTTP (the use of a small number of connections, multiplexing the HTTP datagram, compressing the header and allowing the server to "push" data pro-actively to the client whilst eventually using UDP to accomplish these) bear the potential to improve performance and, thus, provide a perspective for using cloud storage inside HPC environments. However,



(a) Latency (b) Throughput

(c) Client CPU Consumption (d) Client Memory consumption

Fig. 7: Results for the different versions of the HTTP protocol

in this evaluation, they couldn't show their benefit. As future work, we aim to validate that REST is a performant and efficient alternative to common HPC I/O protocols in an actual HPC scenario.

References

1. Association, I.T.: About Infiniband. `https://www.infinibandta.org/about-infiniband/`, [Online; accessed 29-July-2019]
2. AWS: AWS S3. `https://aws.amazon.com/de/s3/`, [Online; accessed 19-July-2019]
3. Bent, J., Gibson, G., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M., Wingate, M.: Plfs: a checkpoint filesystem for parallel applications. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. p. 21. ACM (2009)
4. Bortolotti, D., Carbone, A., Galli, D., Lax, I., Marconi, U., Peco, G., Perazzini, S., Vagnoni, V.M., Zangoli, M.: Comparison of udp transmission performance between ip-over-infiniband and 10-gigabit ethernet. IEEE Transactions on Nuclear Science **58**(4), 1606–1612 (2011)
5. Borzemski, L., Starczewski, G.: Application of transfer regression to tcp throughput prediction. In: 2009 First Asian Conference on Intelligent Information and Database Systems. pp. 28–33. IEEE (2009)

6. Cao, R., Chi, X., Zhao, Y., Xiao, H., Wang, X., Lu, S.: Sceapi: A unified restful web api for high-performance computing. In: J. Phys. Conf. Ser. vol. 898, p. 092022 (2017)
7. Chang, C.S., Thomas, J.A.: Effective bandwidth in high-speed digital networks. IEEE Journal on Selected areas in Communications **13**(6), 1091–1100 (1995)
8. Chen, S., GalOn, S., Delimitrou, C., Manne, S., Martínez, J.F.: Workload characterization of interactive cloud services on big and small server platforms. In: Workload Characterization (IISWC), 2017 IEEE International Symposium on. pp. 125–134. IEEE (2017)
9. intel cloud: cosbench. `https://github.com/intel-cloud/cosbench`, [Online; accessed 19-July-2019]
10. Cloudflare: Implementation of the QUIC protocol. `https://github.com/cloudflare/quiche`, [Online; accessed 01-April-2020]
11. Denis, A., Trahay, F.: Mpi overlap: Benchmark and analysis. In: 2016 45th International Conference on Parallel Processing (ICPP). pp. 258–267. IEEE (2016)
12. Devresse, A., Furano, F.: Efficient http based i/o on very large datasets for high performance computing with the libdavix library. In: Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware. pp. 194–205. Springer (2014)
13. Dillon, T., Wu, C., Chang, E.: Cloud computing: issues and challenges. In: Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on. pp. 27–33. Ieee (2010)
14. DKRZ: Mistral. `https://www.dkrz.de/up/systems/mistral/configuration`, [Online; accessed 19-July-2019]
15. Dumazet, E.: Increase loopback mtu. `https://bit.ly/3c4PHVO` (2012), [Online; accessed 24-Feb-2020]
16. Eitzinger, J., Röhl, T., Hager, G., Wellein, G.: Likwid 4 tools architecture
17. Evangelinos, C., Hill, C.: Cloud computing for parallel scientific hpc applications: Feasibility of running coupled atmosphere-ocean climate models on amazons ec2. ratio **2**(2.40), 2–34 (2008)
18. Expósito, R.R., Taboada, G.L., Ramos, S., Touriño, J., Doallo, R.: Performance analysis of hpc applications in the cloud. Future Generation Computer Systems **29**(1), 218–229 (2013)
19. Folk, M., Heber, G., Koziol, Q., Pourmal, E., Robinson, D.: An overview of the hdf5 technology suite and its applications. In: Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases. pp. 36–47. ACM (2011)
20. Gettys, J.: SMUX Protocol Specification . `https://www.w3.org/TR/1998/WD-mux-19980710` (1998), [Online; accessed 19-July-2019]
21. Glozer, W.: wrk - a HTTP benchmarking tool. `https://github.com/wg/wrk`, [Online; accessed 19-July-2019]
22. Goodell, D., Kim, S.J., Latham, R., Kandemir, M., Ross, R.: An evolutionary path to object storage access. In: 2012 SC Companion: High Performance Computing, Networking Storage and Analysis. pp. 36–41. IEEE (2012)
23. Grant, R.E., Balaji, P., Afsahi, A.: A study of hardware assisted ip over infiniband and its impact on enterprise data center performance. In: 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS). pp. 144–153. IEEE (2010)
24. Gruber, T.: Likwid:about L3 evict. `https://github.com/RRZE-HPC/likwid/issues/213`, [Online; accessed 13-July-2019]
25. Gupta, A., Sarood, O., Kale, L.V., Milojicic, D.: Improving hpc application performance in cloud through dynamic load balancing. In: Cluster, Cloud and Grid

Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on. pp. 402–409. IEEE (2013)
26. h2load: benchmarking tool for HTTP/2 server. https://nghttp2.org/documentation/h2load.1.html, [Online; accessed 19-October-2019]
27. He, Q., Dovrolis, C., Ammar, M.: On the predictability of large transfer tcp throughput. Computer Networks **51**(14), 3959–3977 (2007)
28. Huppler, K.: The art of building a good benchmark. In: Technology Conference on Performance Evaluation and Benchmarking. pp. 18–30. Springer (2009)
29. IETF: QUIC Working Group. https://quicwg.org/, [Online; accessed 01-April-2020]
30. IETF: Request for Comments: 6298 . https://tools.ietf.org/html/rfc6298 (2011), [Online; accessed 19-January-2020]
31. Intel: Adress Translation on Intel X56xx. https://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring/topic/277182, [Online; accessed 15-September-2019]
32. Intel: An Introduction to the Intel® QuickPath Interconnect. https://www.intel.com/technology/quickpath/introduction.pdf, [Online; accessed 15-September-2019]
33. Intel: Intel® Xeon® Processor E5-2680. https://ark.intel.com/content/www/us/en/ark/products/81908/intel-xeon-processor-e5-2680-v3-30m-cache-2-50-ghz.html, [Online; accessed 15-September-2019]
34. Kneschke, J.: lighttpd. https://www.lighttpd.net/, [Online; accessed 29-July-2019]
35. Ko, R.K., Kirchberg, M., Lee, B.S., Chew, E.: Overcoming large data transfer bottlenecks in restful service orchestrations. In: 2012 IEEE 19th International Conference on Web Services. pp. 654–656. IEEE (2012)
36. Kunkel, J.: md-workbench. https://github.com/JulianKunkel/md-workbench, [Online; accessed 19-August-2019]
37. Lafayette, L.: Exploring issues in event-based hpc cloudbursting
38. Liu, J., Koziol, Q., Butler, G.F., Fortner, N., Chaarawi, M., Tang, H., Byna, S., Lockwood, G.K., Cheema, R., Kallback-Rose, K.A., et al.: Evaluation of hpc application i/o on object storage systems. In: 2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS). pp. 24–34. IEEE (2018)
39. Liu, J., Chandrasekaran, B., Yu, W., Wu, J., Buntinas, D., Kini, S., Panda, D.K., Wyckoff, P.: Microbenchmark performance comparison of high-speed cluster interconnects. Ieee Micro **24**(1), 42–51 (2004)
40. Lofstead, J., Jimenez, I., Maltzahn, C., Koziol, Q., Bent, J., Barton, E.: Daos and friends: a proposal for an exascale storage system. In: SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 585–596. IEEE (2016)
41. Ma, D., Zhang, W., Li, Q.: Dynamic scheduling algorithm for parallel real-time jobs in heterogeneous system. In: Computer and Information Technology, 2004. CIT'04. The Fourth International Conference on. pp. 462–466. IEEE (2004)
42. Mell, P., Grance, T., et al.: The nist definition of cloud computing (2011)
43. Netto, M., N. Calheiros, R., R. Rodrigues, E., Cunha, R., Buyya, R.: Hpc cloud for scientific and business applications: Taxonomy, vision, and research challenges **51** (10 2017)
44. ngtcp2: Effort to implement IETF QUIC protocol. https://github.com/ngtcp2/ngtcp2, [Online; accessed 01-April-2020]

45. NLANR/DAST: Iperf. https://github.com/esnet/iperf, [Online; accessed 11-July-2019]
46. OpenLiteSpeed: OpenLiteSpeed Web Server. https://openlitespeed.org/, [Online; accessed 19-December-2019]
47. OpenSSL: QUIC and OpenSSL. https://www.openssl.org/blog/blog/2020/02/17/QUIC-and-OpenSSL/, [Online; accessed 01-April-2020]
48. Piderit, R., Mainoti, G.: Mitigating user concerns to maximize trust on cloud platforms. In: 2016 IST-Africa Week Conference. pp. 1–9. IEEE (2016)
49. Richardson, L., Ruby, S.: RESTful web services. " O'Reilly Media, Inc." (2008)
50. Tene, G.: A constant throughput, correct latency recording variant of wrk. https://github.com/giltene/wrk2, [Online; accessed 11-July-2019]
51. Thakur, R., Gropp, W., Lusk, E.: Data sieving and collective i/o in romio. In: Proceedings. Frontiers' 99. Seventh Symposium on the Frontiers of Massively Parallel Computation. pp. 182–189. IEEE (1999)
52. The MPI Forum, C.: Mpi: A message passing interface. In: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing. pp. 878–883. Supercomputing '93, ACM, New York, NY, USA (1993). https://doi.org/10.1145/169627.169855, http://doi.acm.org/10.1145/169627.169855
53. Tianhua, L., Hongfeng, Z., Guiran, C., Chuansheng, Z.: The design and implementation of zero-copy for linux. In: 2008 Eighth International Conference on Intelligent Systems Design and Applications. vol. 1, pp. 121–126. IEEE (2008)
54. Truong, H.L., Dustdar, S.: Composable cost estimation and monitoring for computational applications in cloud computing environments. Procedia Computer Science **1**(1), 2175–2184 (2010)
55. Wu, K., Arpaci-Dusseau, A., Arpaci-Dusseau, R.: Towards an unwritten contract of intel optane ssd. In: 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19). USENIX Association, Renton, WA (2019)
56. Zadok, E., Hildebrand, D., Kuenning, G., Smith, K.A.: Posix is dead! long live... errr... what exactly. In: Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems. pp. 12–12. USENIX Association (2017)
57. Zhang, Y., Meisner, D., Mars, J., Tang, L.: Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. In: ACM SIGARCH Computer Architecture News. vol. 44, pp. 456–468. IEEE Press (2016)