Evaluating SciStream (Federated Scientific Data Streaming Architecture) on FABRIC

Chengyi Qu*, Joaquin Chung†, Zhengchun Liu‡, Tekin Bicer‡, and Rajkumar Kettimuthu‡
* University of Missouri, † The University of Chicago, ‡ Argonne National Laboratory
Email: cqy78@mail.missouri.edu, chungmiranda@uchicago.edu, zhengchun.liu@anl.gov,
tbicer@anl.gov, kettimut@mcs.anl.gov

***Abstract*—Modern scientific workflows that require data reduction, feature detection, and experiment steering in real time are becoming more common nowadays. However, efficient and secure data streaming is challenging to realize in practice, because of a lack of direct external network connectivity for scientific instruments and because of authentication and security requirements of HPC centers. The `SciStream` toolkit was proposed as a middlebox-based architecture with control protocols to enable efficient and secure memory-to-memory data streaming between producers and consumers that lack direct network connectivity. However, an initial evaluation uncovered that a naive implementation of `SciStream` may add unwanted jitter to a scientific streaming pipeline. Thus, in this paper we present implementation approaches for `SciStream`, and we evaluate those implementations on the FABRIC national testbed. Furthermore, we present the first attempt to develop a streaming application using the QUIC transport protocol and we evaluate it over a `SciStream`-enabled real WAN on FABRIC.**

***Index Terms*—Scientific streaming, SciStream, S2DS**

I. Introduction

As scientific instruments generate data at rates in the order of tens of gigabytes per second [1], real-time analysis of streaming data has emerged as a solution to cope with this new paradigm [2]–[5]. Furthermore, rapid analysis of generated data may permit real-time feedback and experiment steering, however this often requires computational capabilities greater than those available at a single experimental facility—and/or may require the use of specialized computer systems [6]. Thus, the use of remote high-performance computing (HPC) for analysis is becoming more common in large scientific streaming workflows.

We note that scientific streaming applications differ from many other streaming applications in their high throughput requirements (a single application may require $>$10 Gbps) and the fact that the data producers (e.g., data acquisition applications on scientific instruments, simulations on supercomputers) and consumers (e.g., data analysis applications on HPC systems) are typically in different security domains (and thus require bridging of those domains). The `SciStream` system described in [7] (a brief overview is provided in the next section) establishes the necessary bridging and end-to-end authentication between source and destination, while enabling efficient memory-to-memory data streaming. However, in our preliminary evaluations, we discovered that a naive implementation of `SciStream`'s Data Server (`S2DS`), can introduce significant jitter into an end-to-end streaming pipeline. Moreover, measurements obtained from a real testbed were different from those obtained on an emulated scenario.

To get `SciStream` ready for production environments, we first need to evaluate it at scale. Fortunately, the FABRIC [8] national testbed provides a suitable environment to test `SciStream` at scale. With a Terabit network supercore and coast-to-coast 100 Gbps core, FABRIC provides unprecedented programmability along with large amounts of compute and storage built into the core of the network. FABRIC uses specially designed switching nodes into the footprint of the Department of Energy's Energy Sciences Network (ESnet) and regional networks across the United States.

In this paper we present several implementation approaches for `SciStream`'s `S2DS` that may alleviate the large jitter introduced by our initial prototype, and we evaluate their performance on FABRIC under different WAN setups. Furthermore, we present an early prototype of an streaming application using the QUIC transport protocol over a `SciStream`-enabled infrastructure on FABRIC.

The rest of the paper is organized as follows. We provide background and motivation in Section II. We describe implementation approaches for `SciStream`'s `S2DS` in Section III, and we present our evaluation results in Section IV. We conclude in §V with a brief summary and look at future work.

II. Background and Motivation

`SciStream` [7] is a federated system in which participating scientific facilities (typically in independent administrative domains) make their resources available through programmatic interfaces to enable wide-area streaming analysis. The `SciStream` architecture relies on gateway nodes (GNs) and control protocols to create *on-demand proxies* between an instrument's LAN and the WAN as shown in Figure 1. The reasoning behind `SciStream` use proxies at the transport layer ensures that the architecture is agnostic of streaming application libraries and implementations. `SciStream` has

This material was based upon work supported by the U.S. National Science Foundation (NSF), under award 2019073. It was supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

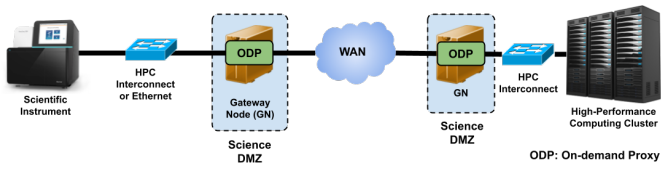

Fig. 1: SciStream architecture for supporting efficient and secure data streaming from data producer's memory to remote data consumer's memory using gateway nodes and on-demand proxies (ODP) [7].

three software components that participate in the control protocols:

- **SciStream User Client (S2UC)** is a software with which the end user and/or workflow engines/tools acting on behalf of the user interact to orchestrate end-to-end data streaming.
- **SciStream Control Server (S2CS)** runs on a GN to manage resources, including initiating, monitoring, and terminating on-demand proxy processes.
- **SciStream Data Server (S2DS)** runs on gateway nodes and acts as a proxy between the internal network (LAN or HPC interconnect) and the external WAN.

FABRIC [8] is a suitable infrastructure for evaluating the efficiency of scientific data streaming enabled by SciStream from the perspective of throughput achieved and latency introduced. Currently, FABRIC has many sites connected at 100 Gbps, and experimenters can create network topologies various values of RTT ranging from LAN environments (<1 ms) to continental WAN (>100 ms). Existing testbeds may fall short on different requirements for the type of evaluations we want to perform on SciStream's S2DS. Those that can provide a 100 Gbps WAN link (e.g., Chameleon Cloud [9]) may lag the diversity of many sites with various RTT values, while those with many sites do not provide high capacity in the WAN (e.g., GENI [10]). In this paper we evaluate several implementations of SciStream's S2DS over different FABRIC setups. The following sections describe our implementation approaches (Section III) and evaluation results (Section IV).

III. Implementation Approaches

In a SciStream-enabled environment, each end-to-end data stream must traverse three network segments—data producer to local S2DS, local S2DS to remote S2DS, and remote S2DS to data consumer. As SciStream is designed to work at the transport layer, in this section we describe implementation approaches for S2DS using both TCP and QUIC transport protocols.

A. Simple TCP Proxy Implementation

The simplest implementation of S2DS is a TCP port forwarder. For our SciStream prototype described in [7], we added a reconfigurable circular buffer. In this port forwarding strategy, GNs listen on a local TCP port and send all received data to a remote host that can either be another GN or a consumer application. This operation utilizes the traditional listen, bind, and send functions of socket programming. These functions utilize user space buffers to queue large amounts of data (see Figure 2 option 1). Although this approach could handle various types of scientific streaming application, it introduced a significant amount of jitter. We attribute the high jitter to variance in context switch and data copy latency between kernel and user space. High jitter can have negative impact on streaming applications.

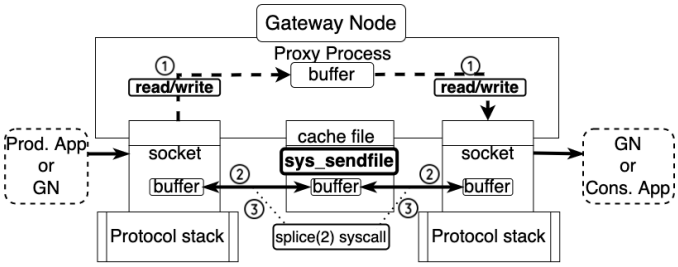

Fig. 2: TCP proxy implementations using the socket programming model: 1) user space buffer, 2) *sendfile*, and 3) *splice(2)*

To reduce jitter, we investigated a few kernel operations and functions to reduce the frequent memory calls. In Linux systems specifically, we identified two specialized syscalls that aim to address this problem: $sendfile$ and $splice(2)$. The $sendfile$ syscall can be used to speed up transferring large files from disk to a socket (see Figure 2 option 2). However, this approach keeps the copying operation between two sockets that increases the load on the proxy as the streaming application load increases. On the other hand, the Linux $splice(2)$ function (see Figure 2 option 3) avoids copying data from kernel to user space, however it requests syscalls for each forwarded packet, increasing the operation cost.

In summary, processes that forward large amounts of data between sockets face three problems:

(i) Making multiple *syscalls* for every forwarded packet is costly.
(ii) The user space process must be woken up often to forward the data. Depending on the scheduler, this may result in poor tail latency.
(iii) Copying data from kernel to user space and then immediately back to the kernel is not free and accumulates to a measurable cost.

Thus, in the next section we present a new approach to building a TCP proxy that circumvent these problems.

B. High-Speed TCP Splicing Proxy using SOCKMAP

In this section we provide an alternative implementation of a TCP proxy that could potentially reduce the context switching between kernel and user space, which may translate in improved jitter performance for scientific streaming applications. By leveraging the Linux eBPF (Berkeley Packet Filter) technology [11], we can build a TCP proxy in which all packet operations will be processed in kernel space only. eBPF is a technology used to run sandboxed programs in kernel space, without requiring to change the kernel's source code.

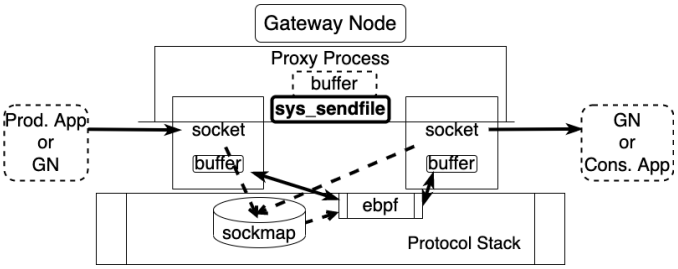

Fig. 3: High-Speed TCP Splicing Proxy using SOCKMAP.

In the traditional socket programming model, a socket opens an interface between kernel TCP/IP stack and a task context in user space. Then, the socket forwards data from the kernel space to the tasks where the socket is binding. To simplify this, eBPF introduces *sockmap* [12], an API that utilizes a method called *Stream Parser* to pass packet to other eBPF tasks. These eBPF tasks can achieve a streaming redirection operation on kernel space only, this direct operation will speed up the data transmission compared to the original socket programming. Figure 3 shows the basic logic on how *sockmap* is applied in one GN. Instead of directly loading processes on user space or communicating with a cache file, TCP socket descriptors in a GN are now inserted into the $SOCKMAP$, or specifically $BPF\_MAP\_TYPE\_SOCKMAP$, as a type of an eBPF map. After the insertion, eBPF will handle the forwarding procedure, which ensures the processing happens only in kernel space instead of the user space.

Algorithm 1 Sockmap TCP Proxy

Require: $sockmap$, $proxymap$, $[IPs, ports]$
  initialization $socket1$, $socket2$, ...
  for all $index, [IP, port] \in [IPs, ports])$ **do**
    bind $[socket]$ with $[IP, port]$
    $sockmap[index] \leftarrow [socket, index]$
  end for
  while $True$ **do**
    $congest\_ctrl()$ & $buf.push(sockmap[index])$
    $proxymap[target\_port] \leftarrow buf.pop(sockmap[index])$
  end while

Algorithm 1 shows the pseudocode of a TCP proxy that uses the *sockmap* function according to the eBPF specifications. The $sockmap$ and $proxymap$ are defined as eBPF map structures with the type of $SOCKMAP$ and $HASH$, respectively.

C. Implementation of a QUIC Proxy for Secure Streaming

QUIC [13] is a new transport protocol standardized by the IETF, and developed on top of UDP with the aim of being encrypted by default. Furthermore, the HTTP/3 [14] protocol has been developed by the IETF as an application layer protocol on top of QUIC. In this subsection, we present our `SciStream` QUIC proxy solution that remains at the transport layer using QUIC stream mode. Unlike the datagram mode used for video or game streaming purposes, steam mode provides reliable data streaming.

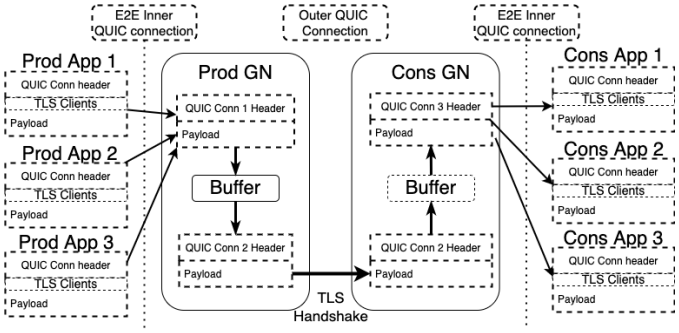

Fig. 4: QUIC proxy implementation with encryption streaming by default.

To develop a general QUIC proxy, we first tried to implement a plain UDP proxy. However, due the features of the QUIC standard [15], an address validation is periodically performed either on the long datagram header or short QUIC header. As a result, the QUIC handshake procedure fails when passing across a plain UDP proxy. To overcome this issue, we implemented a QUIC proxy that creates two separate QUIC connections at each GN: one for LAN and one for WAN. In our QUIC proxy, streaming data will be buffered at the system level inside the proxy node itself and forwarded to the pre-assigned target. Specifically, two GNs first establish a secure QUIC connection on WAN and remain idle waiting for connection requests from the application side. Once a producer or consumer application initiates a connection to their corresponding GN, a new QUIC connection is established and data is forwarded to the target node. However, QUIC transfers over high-bandwidth links can be limited by the size of the system default UDP receive buffer. This buffer holds packets that have been received by the kernel, but have not yet been read by the application. For high-throughput streaming applications, a small buffer size may cause a significant packet drop or even stop the connection. To solve this limitation, before instantiating the QUIC proxy, we must first increase the UDP buffer size in the operating system. Figure 4 shows the end-to-end data flow of our `SciStream` QUIC proxy implementation.

IV. Evaluation

In this section we evaluate the implementation approaches for `SciStream`'s `S2DS` described in Section III on the FABRIC testbed. For our experiments, we are interested in evaluating scientific data streaming over different WAN configurations.

A. Experimental Setup

Figure 5 shows the logical topology of our experimental setup, as drawn by FABRIC's GUI. In general, we request FABRIC resources from two sites and connect them via a WAN link and a separate control network. For the example in Figure 5, we are using Salt Lake City (SALT) and Utah (UTAH) sites. Inside each site, we request at least two compute nodes that are connected by a LAN link (`prod_lan_net`

and cons_lan_net in the topology diagram). Two of these nodes are also connected to the WAN (GN_WAN_NET in the case of the illustration). We also need an out-of-band network for running the SciStream control protocol (control_net). Note that one site (UTAH) has three compute nodes, we use this third node to execute SciStream's S2UC for the control protocol. Each compute node runs Linux Ubuntu 20.04 as the OS, with Linux source code version 5.4.0-97 for eBPF support. Python 3.8 is used for emulating the data generation and consumption processes, and ZeroMQ [16] is used for implementing a Pub/Sub streaming application pipeline in Python. We use HTCP as the congestion control in all our nodes, and tune the TCP setting for WAN scenarios. Our QUIC proxy implementation is based on GoLang version 1.18.3 and quic-go version 0.28.0.

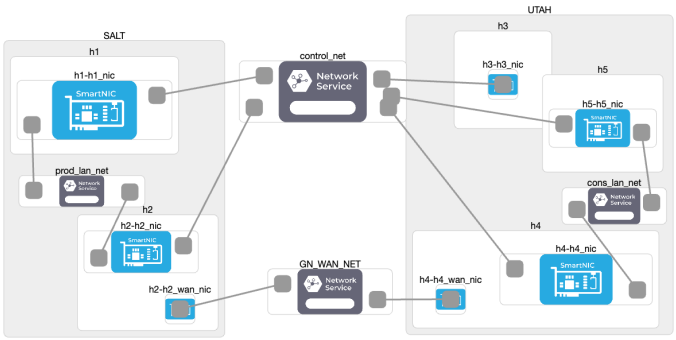

Fig. 5: Experimental setup topology on FABRIC

FABRIC provides granularity for reserving even the type of network interface card (NIC) per compute node. For our experiments, the nodes running producer and consumer applications on each site have one ConnectX-6 Dx 100GbE dual Ethernet NIC card for the following purposes: i) establish a link to the control network and ii) connect to their local GN for data streaming. On the other hand, each GN has three basic 100GbE single NIC cards. The first two NICs are used for the same purpose as indicated for the nodes running the producer and consumer application, while the third NIC card is used for establishing a WAN link between remote GNs.

1) Methodology: To evaluate the performance of S2DS implementation approaches on different WAN settings, we compare the application goodput of SciStream with a scenario in which producer and consumer have direct connectivity over the network (we call this scenario "No-SciStream"). We also evaluate how different implementations of S2DS affect the jitter of a scientific streaming analysis pipeline.

We conduct our evaluation on five setups with RTTs representative of scientific streaming scenarios (see Table I for details). We chose a LAN setup between FABRIC's Utah and Salt Lake City sites ($RTT < 1\ ms$), a metro setup between Michigan and StarLight in Chicago ($RTT < 10\ ms$), a short WAN between StarLight and Dallas sites ($RTT \sim 25\ ms$), a WAN between Utah and the MAX Gigapop in Maryland ($RTT < 60\ ms$), and a long WAN between Michigan and TACC ($RTT < 150\ ms$). We assume that cross-traffic from other experimenters using FABRIC is present in the network.

TABLE I: Avg. RTT (ms) for various FABRIC WAN setups.

	LAN	Metro	Short WAN	WAN	Long WAN
Prod LAN	0.087	0.092	0.167	0.165	0.161
Cons LAN	0.100	0.105	0.179	0.148	0.160
GN WAN	0.253	5.293	23.998	57.848	143.370
Overall	**0.440**	**5.490**	**24.344**	**58.161**	**143.691**

For the all experiments, the producer generates samples as fast as possible (i.e., no sampling delay). In each experiment we transfer 10 GB for a fixed sample size, which we vary across experiments from 512 bytes to 1 MB in power-of-two increments. For each experiment we compute goodput as the size of the received dataset ($num\_samples \times sample\_size$) divided by the elapsed time between the arrival of the first sample and the arrival of the STOP message ($\Delta\ t = t_{stop} - t_{first\_sample}$). We compute the application latency and jitter by subtracting the arrival time of a message to the consumer minus the arrival time of the previous message ($t_{i+1} - t_i$) and store the result in an array. At the end of the experiment we compute the average and standard deviation of our measurements to obtain the latency and jitter, respectively. We configure the circular buffer of the Legacy S2DS to be 10 MB for all experiments, corresponding to 10 messages for the largest sample size.

B. Results

Figure 6 shows the goodput performance results for our two TCP implementations of SciStream S2DS (legacy and *sockmap* proxies) compared to the No-SciStream case, over the five FABRIC setups described in Section IV-A1. To saturate the 100 Gbps WAN links, we simultaneously run five Pub/Sub processes (paired with five instances of S2DS). In the best case scenario, the maximum throughput that a streaming application with direct connectivity (No-SciStream) can achieve is ~78 Gbps. For the LAN, metro, and short WAN scenarios (Figures 6a, 6b, and 6c), we observe that no significant difference exist between the No-SciStream scheme and any of the S2DS implementations for small sample sizes. On the contrary, for large sample sizes, the *sockmap* approach provider higher goodput than the legacy proxy approach. Furthermore, in the case of the metro setup, *sockmap* is able to even outperform the No-SciStream scheme. For both WAN and long WAN setups, the legacy proxy achieves higher goodput for very large sample sizes ($> 256KB$).

Due to space constraints, we have omitted the application latency results. However, we can comment that for legacy and *sockmap* implementations, the application perceived by the consumer application remains in the order of hundreds of microseconds. The jitter performance results for all TCP implementations are shown in Figure 7. In general, we observe that the average jitter remains relatively constant regardless of the RTT of the setup, with average values under 200 μs. The only exception is the Metro setup, in which average jitter values for both legacy and *sockmap* proxies are almost doubled. This can be attributed to a transient event in the

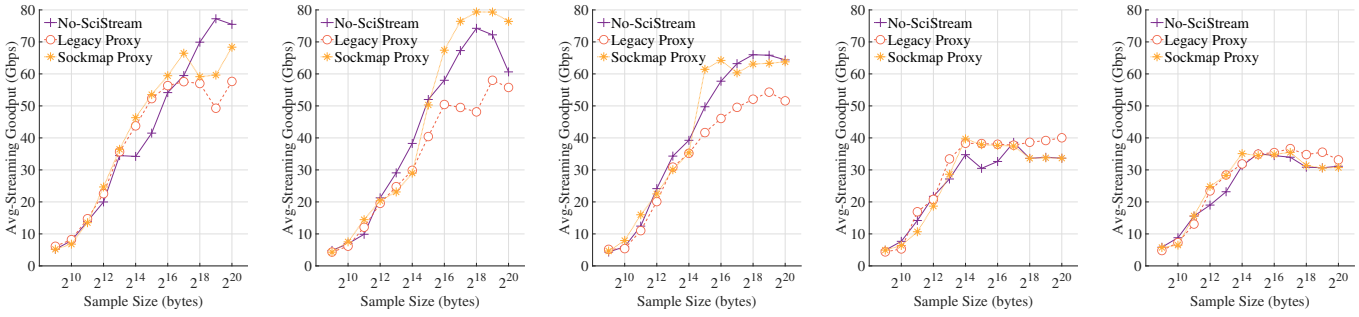

Fig. 6: Goodput performance of two TCP implementations of `SciStream S2DS` (legacy and *sockmap* proxies) compared to the No-SciStream case over five network setups: (a) LAN, (b) metro, (c) short WAN, (d) WAN, and (e) long WAN.

network. For future work, we will increase the repetition of the experiments to reduce the effects of network transients. For all network setups (except for the Metro case), any implementation of `S2DS` provides a smaller jitter on average compared to the No-SciStream configuration. This is explained by the fact that `SciStream` breaks the end-to-end connection into three connections with smaller RTT and it also add buffers in the path.

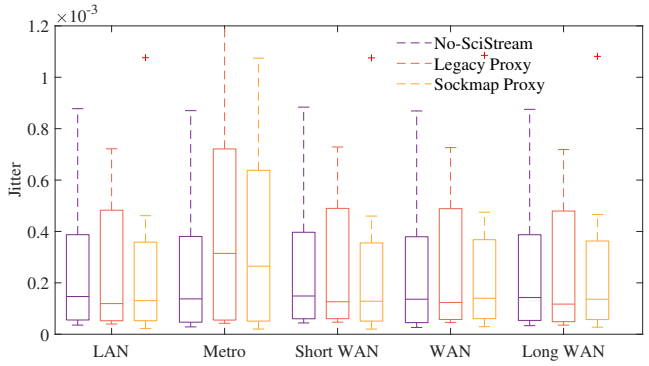

Fig. 7: Jitter performance of two TCP implementations of `SciStream S2DS` (legacy and *sockmap* proxies) compared to the No-SciStream case.

Figure 8 shows the goodput evaluation results for our QUIC proxy implementation. We determined that by default, QUIC assumes a minimum IP packet size of 1280 bytes. However, for all of our experiments, we configure jumbo frames (9000 bytes) in all links of our network topology. Although the maximum goodput in the LAN is ∼22 Gbps and the goodput decreases as the RTT in the WAN increases, we observe that our QUIC proxy implementation provides a goodput comparable to the No-SciStream case. In the case of jitter results (see Figure 9), we observe lower jitter for our QUIC proxy compared to the No-SciStream case. Compared to the TCP case, both application latency (data not shown) and jitter for QUIC are an order of magnitude larger.

V. Conclusion

In this paper we evaluated several implementation approaches of `SciStream`'s `S2DS` over the FABRIC national testbed, evaluating scientific data streaming over different WAN configurations. We compared the application goodput of

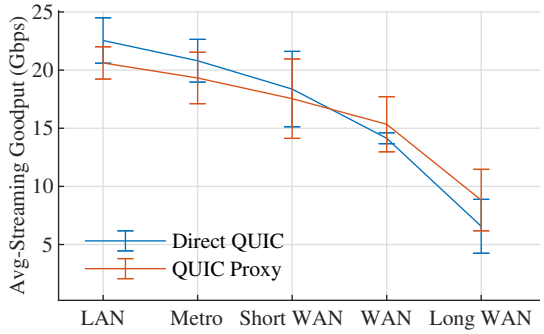

Fig. 8: Average goodput performance of a QUIC application with direct connectivity and over a `SciStream`-enabled QUIC proxy.

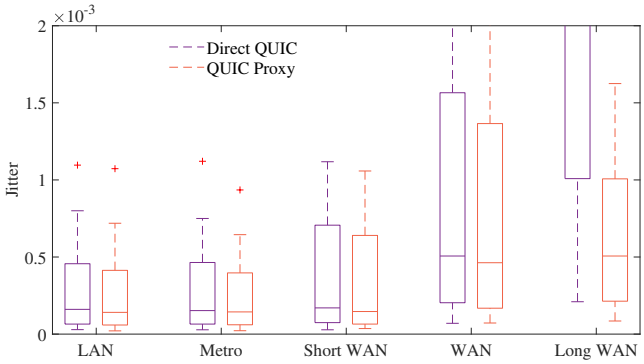

Fig. 9: Jitter performance of a QUIC application with direct connectivity and over a `SciStream`-enabled QUIC proxy.

`SciStream` with a scenario in which producer and consumer have direct connectivity over the network, and we observed no significant difference exist between any implementation of `S2DS` and the direct connectivity case. We also evaluated how different implementations of `S2DS` affect the jitter of a scientific streaming analysis pipeline, and we demonstrated that the *sockmap* implementation provides lower jitter in comparison to a legacy implementation based on socket programming model. Finally, we presented an early prototype of an streaming application using the QUIC transport protocol over a `SciStream`-enabled infrastructure. For our future work, we will continue improving the performance of `SciStream` using FABRIC.

REFERENCES

[1] DOE-SC, "The Report of the BES Advisory Subcommittee on Future X-ray Light Sources," https://science.osti.gov/-/media/bes/besac/pdf/Reports/Future_Light_Sources_report_BESAC_approved_72513.pdf, 2013, accessed: 2021-09-06.

[2] NSF's Big Ideas, "Harnessing data for 21st century science and engineering," https://www.nsf.gov/news/special_reports/big_ideas/harnessing.jsp, 2017, Accessed: 2020-01-06.

[3] Advanced Scientific Computing Research and Basic Energy Sciences, U.S. Department of Energy, "Exascale Requirements Review," https://www.osti.gov/servlets/purl/1341721, 2015, Accessed: 2020-01-06.

[4] A. L. Kinney and D. M. Tilbury, "Dear Colleague Letter: Data-Driven Discovery Science in Chemistry (D3SC)," https://www.nsf.gov/pubs/2018/nsf18075/nsf18075.pdf, 2018, Accessed: 2021-03-01.

[5] NSF, "Transforming science through cyberinfrastructure: NSF's blueprint for a national cyberinfrastructure ecosystem for science and engineering in the 21st century."

[6] R. Kettimuthu, Z. Liu, D. Wheeler, I. Foster, K. Heitmann, and F. Cappello, "Transferring a petabyte in a day," *4th International Workshop on Innovating the Network for Data Intensive Science (INDIS) 2017*, pp. 1–11, Nov. 2017.

[7] J. Chung, W. Zacherek, A. Wisniewski, Z. Liu, T. Bicer, R. Kettimuthu, and I. Foster, "Scistream: Architecture and toolkit for data streaming between federated science instruments," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 185–198. [Online]. Available: https://doi.org/10.1145/3502181.3531475

[8] I. Baldin, A. Nikolich, J. Griffioen, I. I. S. Monga, K.-C. Wang, T. Lehman, and P. Ruth, "Fabric: A national-scale programmable experimental network infrastructure," *IEEE Internet Computing*, vol. 23, no. 6, pp. 38–47, 2019.

[9] K. Keahey, P. Riteau, D. Stanzione, T. Cockerill, J. Manbretti, P. Rad, and R. Paul, "Chameleon: a scalable production testbed for computer science research," *Contemporary High Performance Computing*, vol. 3, 2017.

[10] C. Elliott, "Geni-global environment for network innovations." in *LCN*, 2008, p. 8.

[11] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Júnior, and L. F. M. Vieira, "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications," *ACM Comput. Surv.*, vol. 53, no. 1, feb 2020. [Online]. Available: https://doi.org/10.1145/3371038

[12] J. Sitnicki, "Steering connections to sockets with bpf socket lookup hook," 2020, eBPF Summit. [Online]. Available: \url{https://ebpf.io/summit-2020-slides/eBPF_Summit_2020-Lightning-Jakub_Sitnicki-Steering_connections_to_sockets_with_BPF_socke_lookup_hook.pdf}

[13] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi, "The quic transport protocol: Design and internet-scale deployment," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 183–196. [Online]. Available: https://doi.org/10.1145/3098822.3098842

[14] G. Perna, M. Trevisan, D. Giordano, and I. Drago, "A first look at http/3 adoption and performance," *Computer Communications*, vol. 187, pp. 115–124, 2022.

[15] J. Iyengar and M. Thomson, "Rfc 9000 quic: A udp-based multiplexed and secure transport," *Omtermet Emgomeeromg Task Force*, 2021.

[16] P. Hintjens, *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.", 2013.

LICENSE