Shimmy: Accelerating inter-container communication for the IoT Edge

Manan Khasgiwale*, Vasu Sharma*, Shivakant Mishra, Biljith Thadichi
Department of Computer Science, *University of Colorado* Boulder, USA
{Manan.Khasgiwale|Vasu.Sharma|mishras|Biljith.Thadichi}@colorado.edu

Jaiber John and Rahul Khanna
Intel Corporation, Oregon, USA
{Jaiber.J.John|Rahul.Khanna}@intel.com

GLOBECOM 2023 - 2023 IEEE Global Communications Conference | 979-8-3503-1090-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/GLOBECOM54140.2023.10437823

***Abstract*— Cloud-native technologies consisting of containers, microservices, and service meshes bring the traditional advantages of Cloud Computing like scalability, composability, and rapid deployability to the IoT Edge. An application built on the microservices architecture relies on a collection of individual containerized components offering modular services via REST or gRPC interfaces over the network. Compared to a monolithic application, the magnitude of data and control exchange between the components of a microservices application is several orders higher. Studies have shown that overheads caused by such inter-container communication are a significant hurdle in achieving the sub-50ms latencies required for 5G enabled network Edges comprised of a much smaller compute cluster, unlike the Cloud. In this paper, we present Shimmy - a shared memory-based communication interface for containers that is cleanly integrated into the Kubernetes orchestration architecture while offering significant acceleration for microservices. Results have shown a consistent 3-4x latency improvement over UDP and TCP, as much as 20x latency improvement over RabbitMQ, while significantly reducing memory and CPU usage in large data transfers as well as real-time video streaming.**

***Index Terms*—microservice, IoT, Cloud-native, intercontainer latencymicroservice, IoT, Cloud-native, intercontainer latency**

I. Introduction

A *microservice-based architecture* is commonly used to structure complex applications at the Edge. In this architecture, an application is built from a selection of individually isolated microservices, where each microservice is a self-contained piece of software with clearly-defined interfaces implementing a simple functionality [1]. Containers provide a very convenient way to implement microservices. However, they isolate their functionality in their own compute environment resulting in forming information barriers and raise a need for inter-service communication over a (virtual) network. This results in higher costs in terms of network stack latency and message processing time. Indeed, recent performance studies conducted at Google and Facebook data centers have revealed that the CPU cycles consumed by operations that are not part of the application logic can be as high as 80% of the total compute times and a majority of that overhead comes from a need for moving data between different microservices [2], [3].

In this paper, we address the critical issue of inter-container communication performance in building a microservices-based application. In particular, we propose that we exploit shared memory communication channels and treat interconnected containers as being connected via a collection of shared memory channels that can be either bi-directional streams (as in TCP/IP) or publish/subscribe channels. This would not only optimize communication when containers are co-located on the same server but can also support fast and efficient remote communication by synchronizing memory regions via Remote Direct Memory Access (RDMA) technology [4].

We propose Shimmy, a shared memory-based inter-container communication mechanism. For containers running on the server, Shimmy creates shared memory that the containers can write into and read from, and for containers running on different servers, Shimmy utilizes RDMA.

We have implemented a prototype of Shimmy that is integrated with Kubernetes. A detailed performance evaluation demomstrates thatShimmy provides significant improvement in communication latency over UDP, TCP, and RabbitMQ for both packet-based and streaming communication while significantly reducing the overall resource utilization. The major findings of this paper are as follows:

- Shimmy provides significant performance improvement in communication latency for both packet-based and streaming applications.
- Shimmy significantly reduces the overall CPU and memory usage, which is particularly useful for relatively less powerful servers at the edge when compared to the cloud.
- Shimmy has been integrated with Kubernetes, one of the most popular container orchestration frameworks. As a result, users may continue to take advantage of all the wonderful features of Kubernetes while availing the performance advantages of Shimmy without having to modify their code in any significant way.
- A prototype of Shimmy has been implemented and we plan to release it as open-source to the researchers and developers to use and extend.

II. Related Work

Various researchers have investigated and provided solutions for the problem of optimizing inter-container networking given its significance in achieving optimal Edge performance. Ubaid et al. [6] have analyzed the inter-container network bandwidth and compute utilization on an RDMA-enabled Kubernetes cluster, comparing various networking frameworks and observed significant advantages over traditional networks for

live migration. Microsoft Freeflow [7] provides efficient inter-container networking through virtual RDMA, a pure software implementation over commodity RDMA NICs achieving near bare-metal performance. Xue et al. [16] have shown how deep neural networks can greatly benefit from using an RDMA based system rather than the traditional systems which rely on gRPC on TCP. Fent et al. [17] have extensively compared TCP with a RDMA or a shared memory based solution for a database management system. This work shows that RDMA based solutions provide an order of magnitude better performance than traditional TCP approach. The paper also shows that this performance improvement comes due to lesser context switches in the kernel.

Kun et al. [9] has published a comprehensive comparison of various container networking technologies, analyzing their performance degradation and overheads in a Cloud environment. A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. Gerald et al. [8] have benchmarked various network policies and CNI plugins in Kubernetes and analyzed their applicability for 5G low-latency applications. Tanner et al. [11] have created a software library called Dhmem that manages the shared memory buffer between workflow tasks in separate containers, with minimal code change and performance overhead. Dhmem allows a separate container for each workflow task to be constructed completely independently, allowing easy integration into existing workflow systems. Memif [12] is another open-source implementation of an efficient inter-container shared memory networking library using the Vector Packet Processing (VPP) technology.

Kubernetes [5] is an open-source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery. Pods are the smallest deployable units of computing that you can create and manage in Kubernetes. A Pod is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled and run in a shared context. Inter-container communication is supported via the IP addresses of the Pods. The type of network a container uses for inter-container communication, whether it is a bridge, an overlay, a macVLan network, or a custom network plugin, is transparent from within the container. From the container's point of view, it has a network interface with an IP address, a gateway, a routing table, DNS services, and other networking details.

The idea of using shared memory for inter-container communication was first proposed in [10]. A preliminary prototype demonstrated the potential benefits of this approach. However, the prototype was a stand-alone implementation that would not work with a container orchestration framework. The work presented in this paper is a stable implementation that integrates the idea of using shared memory-based inter-container communication with Kubernetes and provides a detailed performance evaluation.

III. Design and Implementation

A. Design Overview

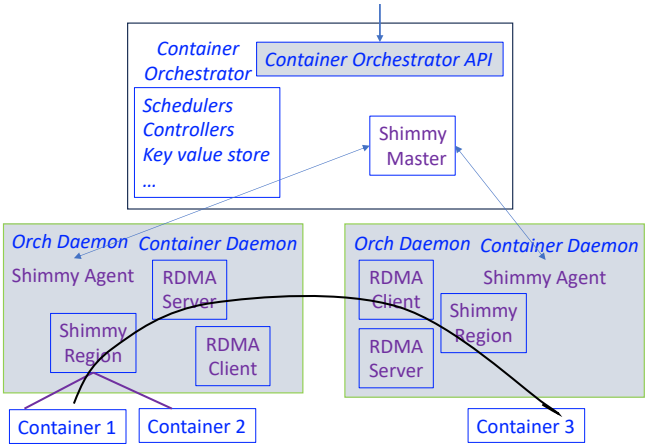

Fig. 1: Shimmy Architecture

Figure 1 shows the overall architecture of Shimmy and how shared memory channels for inter-container communication fit into the overall system. We add two new components to the overall orchestration framework such as Kubernetes: *Shimmy agent* and *Shimmy master*. A single instance of the Shimmy agent runs on each host and shared by all Pods running on that host, and the Shimmy master logically runs as part of the container orchestrator. Logically, a single instance of Shimmy master runs. A system administrator can define new shared memory communication channels as part of the multi-container application. Shimmy master is responsible for tracking container deployment and interfacing with Shimmy agents on each host to ensure that defined communication channels are established between the associated containers (whether local or remote).

In a container orchestration framework such as Kubernetes, a Service is a resource type that provides a single entry point to a group of Pods (containers) running the same application. In Shimmy, we define a new resource type, SharedMemoryObject, that is described through a YAML file which includes (in Kubernetes) a selector (to match against the container names) and a type (pub/sub or stream). When Kubernetes, in this case, deploys a Pod and matches a SharedMemoryObject, the Shimmy master notes this and notifies the Shimmy agent on the host on which the Pod is deployed, and then subsequently attaches a shared memory object of the desired type to the Pod.

B. Lock-Based and Lock-Free Channels

Shimmy is designed to provide two types of communication between containers. In the *Lock-Based approach*, data written to a shared memory channel can be read exactly once. This is similar to a producer-consumer type of communication. On the other hand, in the *Lock-Free approach*, data written to a

shared memory channel remains available to read until either a set number of readers (defined as a parameter) have read that data or until a set timer has expired. This is similar to a publish-subscribe type of communication.

C. Shared Memory Channel - Lock Based Approach

In the Lock Based Approach, the Shimmy agent on each pod is the Broker. It is a RESTful server responsible for creating and deleting two sets of shared memory regions—*data region* to store the data to be communicated, and *metadata region* to store metadata used for managing access to the data region. Data region is structured as a circular buffer that consists of a fixed number of identical blocks. Metadata region includes offsets for the producer and consumer and semaphores. Semaphores facilitate synchronization between the producer and consumer using the standard single-producer and single-consumer synchronization mechanism.

A shared memory channel is identified by a topic-name. The producer and consumer processes need to know the topic-name to start using Shimmy. For each topic-name, a unique identifier for the shared memory as well as its metadata and semaphores are stored in the Redis database.

D. Shared Memory Channel - Lock-Free Approach

A shared memory channel in Lock Free Approach is comprised of three logical shared memory regions—*data region* to store the data to be communicated, *local metadata region* to store each consumer's cycle count and offset, and *global metadata region* to store the cycle count and offset of the producer as well as the offset and cycle count of the slowest consumer. As in the case of Lock-Based approach, the data region in Lock-Free approach is a circular buffer, and the producers and consumers maintain their respective offsets and cycle counts. For each consumer, its offset and cycle count is stored in the local metadata region. Producer offset and its cycle count as well as the offset and cycle count of the slowest consumer is stored in the global metadata region. Unlike Lock-Based approach, no semaphores are used in Lock-Free approach. Instead, cycle counts are used to distinguish between new and stale data at any slot in the circular buffer. A Periodic Scanner updates the slowest consumer offset and cycle count in the global metadata region by scanning through the offsets and cycle counts of all consumers in the local metadata region. This functionality is extended to skip unread data from slow consumers based on a timer.

E. Kubernetes Integration

Shimmy is integrated in Kubernetes using the *Custom Resource Definitions (CRDs)*, an extension of Kubernetes API. While Custom Resources just let you store and retrieve structured data, when combined with a Custom Controller, we get a declarative API that lets us specify the desired state of the cluster, and take care of keeping the current state of the resources in sync with the desired state. So if a user wants to declare a SharedMemoryObject, they can specify the properties of the shared memory in a YAML file just like deployments or services are specified and then follow that with a *kubectl apply -f sharedmemory.yaml*. This request goes to the Custom Controller for shared memory management, responsible for creating the SharedMemoryObject as specified in the YAML file.

F. Shimmy with RDMA

In a scenario where the two communicating containers are hosted on two different servers, Shimmy uses RDMA for memory sharing. Recall that RDMA enables two networked devices to exchange data in main memory without relying on the processor, cache or operating system. In particular, RDMA bypasses the entire network stack. To use RDMA in Shimmy, RDMA server and RDMA client run as part of Shimmy agent. Figure 2 shows the architecture of Shimmy RDMA. We use the RDMA Support to Docker containers provided using virtual RDMA devices (vHCA) implemented using SR-IOV (Single Root I/O Virtualization) capability of the Mellanox ConnectX-4/ConnectX-5 HCAs.

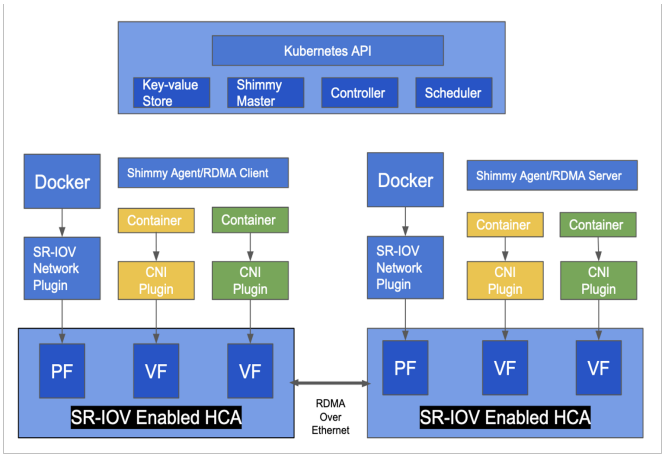

Fig. 2: Shimmy-RDMA Architecture

IV. Results and Performance Analysis

We have done an extensive performance analysis of Shimmy and compared it with other popular solutions for inter-container communication in use today. These include traditional networking protocols (UDP and TCP), broker-based message communication (RabbitMQ), transfer of video files, and real-time streaming video. Performance characteristics that we have measured are communication latency and resource consumption.

Our experimental setup is deployed on Google Cloud and consists of a GKE Standard Kubernetes cluster running on the 1.21.6-gke.1500 version. The cluster node is an e2-standard-4 machine type and consists of four virtual CPUs and 16 GB of RAM. It runs Container-Optimized OS from Google. The allocatable CPU and memory units available to the resources that we create inside this node are 3.92 CPU (3920 milliCPU) and 13.94 GB.

A. Shimmy vs UDP vs TCP

Our first experiment compares communication latency between producer and consumer processes for Shimmy, UDP and TCP. Both the Lock-Based and Lock-Free approaches of Shimmy were used in this experiment. The experiment involved performing 10,000 iterations of sending and receiving data packets varying in sizes from 100 to 65,000 bytes (the maximum packet size for UDP). These data packets are exchanged between the producer and consumer processes. Figure 3 shows per-packet median latencies. Note that these measurements do not include the one-time cost of creating or binding sockets (UDP and TCP), connection establishment (TCP), or setting up a shared memory channel (Shimmy). The readings taken for each of the communication methods in this experiment have been taken in isolation meaning that the producer and consumer processes using that particular communication method were the only processes running on the Kubernetes cluster node.

Our first observation is that both the Lock-Based and Lock-Free approaches of Shimmy consistently outperform UDP and TCP, and they reduce communication latency by as much as 4x. We also notice that the median latency of Shimmy stays somewhat consistent even as the size of the data packets increase, while median latency of UDP and TCP increases with in increase in packet size. The reason for this is that latency in Shimmy is dominated by locking/unlocking of semaphores in Lock-Based approach and by the Periodic Scanner to update slowest consumer's cycle and offset values in the Lock-Free approach. Increase in packet size has relatively lower impact on latency. On the other hand, latency cost in TCP and UDP is dominated by data copying across address spaces. Finally, latency of Lock-Free approach was slightly larger than the latency of Lock-Based approach in our measurement. We observed that the latency of Lock-Free approach critically depends on the the periodic time interval of the Periodic Scanner.

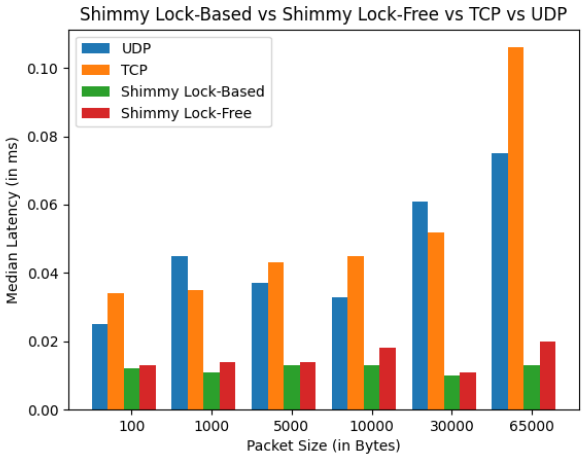

Fig. 3: Latency comparison among Shimmy Lock-Based, Shimmy Lock-Free, TCP and UDP

B. Shimmy vs RabbitMQ

Next, we assess the performance of Shimmy with current state-of-the-art message brokers. Popular message brokers at present include RabbitMQ [14], Apache Kafka [19] and Apache Pulsar [18]. Previous studies have shown that RabbitMQ provides lowest latency when compared to these other message brokers (e.g. See [20], and so we chose to evaluate Shimmy with RabbitMQ. RabbitMQ is one of the most popular open-source message brokers used worldwide by small startups to large enterprises. It facilitates an efficient delivery of messages and is developed around the Advanced Message Queuing Protocol (AMQP) [15], but it is also highly compatible with existing technologies. For larger message sizes (up to 10 MB), we have compared the communication latency of Shimmy lock-based and Shimmy Lock-Free approaches with RabbitMQ. Figure 4 shows the median latency (over 100 iterations) of Shimmy and RabbitMQ for packet sizes ranging from 1,000 bytes to 10 MB (Note that the times shown in this figure are on a log scale). We notice Shimmy (lock-based and lock-free approaches) performs significantly better than RabbitMQ for larger message sizes (roughly 20x lower than RabbitMQ for message sizes above 1 MB). Even for smaller message sizes, it performs around 5-10x better. The high overhead of RabbitMQ can be attributed to the fact that AMQP is an application layer protocol that enables message passing through broker services over TCP/IP connections.

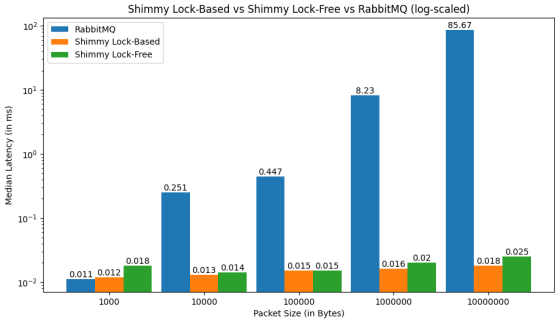

Fig. 4: Latency comparison among Shimmy Lock-Based, Shimmy Lock-free and RabbitMQ (Median latency is reported on a log scale)

C. Resource Usage

Edge servers typically have limited resources in terms of processing and storage capacities when compared to cloud. So, to assess resource usage of Shimmy, we measured the CPU and memory usage over time for multiple parallel communication sessions. These parallel sessions are communication channels between distinct producers and consumers that are spawned concurrently. We compare resource utilization of Shimmy against TCP over these parallel sessions. For Shimmy, the communication channel is a shared memory buffer while for TCP the communication channel is a TCP network socket. Over each of these parallel sessions, data packets of size

10,000 bytes were sent for 10,000 iterations. While doing these measurements, only one set of parallel sessions was active at the time on the cluster node. We used GKE usage metering to capture these utilization metrics for our Google Kubernetes Engine (GKE) cluster node. GKE usage metering can be used to track information about resource requests and actual resource usage of a cluster's workloads.

Figure 5 shows CPU usage for up to twelve parallel sessions for Shimmy and TCP. As we can see from these figures, TCP reached CPU limit of four virtual CPUs that we used in our experimental setup at 12 parallel sessions. On the other hand CPU usage of Shimmy is significantly lower than that of TCP, and remains somewhat consistent with very little increase with increase in the number of parallel sessions, whereas TCP's CPU usage increases dramatically with increase in the number of parallel sessions.

Figure 6 shows memory usage for up to twelve parallel sessions for Shimmy and TCP. Once again, we observe that memory usage of Shimmy is significantly lower than that of TCP, and remains somewhat consistent with very little increase with increase in the number of parallel sessions, whereas TCP's memory usage increases dramatically with increase in the number of parallel sessions.

Figure 5 and 6 report resource usage of Lock-Based approach of Shimmy. Similar trends were observed with the Lock-Free approach of Shimmy as well.

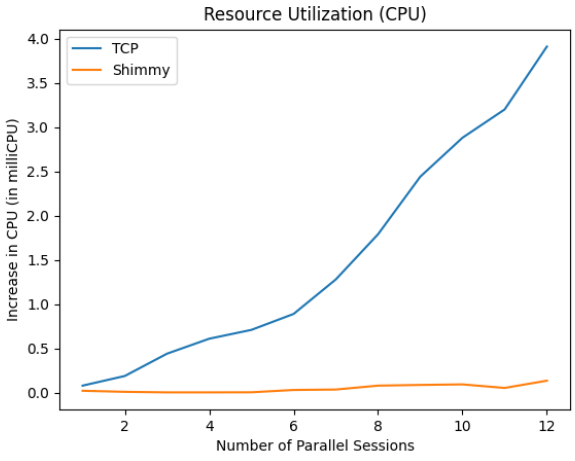

Fig. 5: CPU Usage in Lock-Based Shimmy and TCP

D. Real Time Video Streaming

Finally, we compare resource usage in real-time video streaming using Shimmy vs TCP. We hypothesized that using Shimmy would potentially result in the reduction of the overall resource consumption in comparison to TCP. Video stream capture and segmentation of frames were achieved with the OpenCV [13] library. OpenCV provides a real-time optimized Computer Vision library, tools, and hardware. To transfer a real-time video stream, we first capture and segment the stream into frames in the Producer process and then transfer these

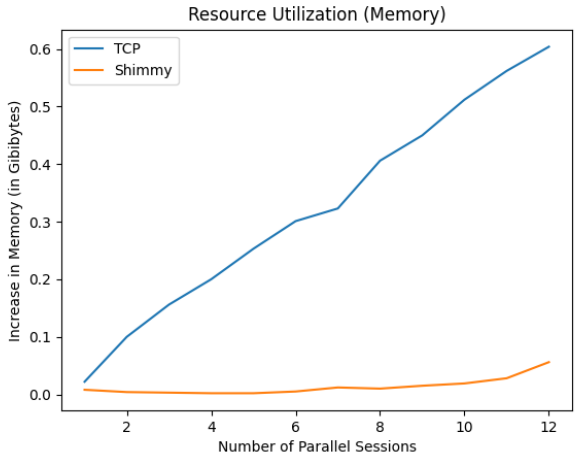

Fig. 6: Memory Usage in Lock-Based Shimmy and TCP

frames to the Consumer process using Shimmy or TCP. The consumer then stitches these frames into a video file. We use the VideoCapture function from OpenCV to capture the stream, the read function to segment the stream into frames, and VideoWriter to reconcile these frames into a video file.

In this experiment we used a publicly available Wowza media stream with H264 video and AAC audio codec as input. This stream has a resolution of 240x160 pixels and a frame rate of 24 fps. This media stream was transferred for over three minutes, and CPU and memory usage metrics were collected during this transfer.

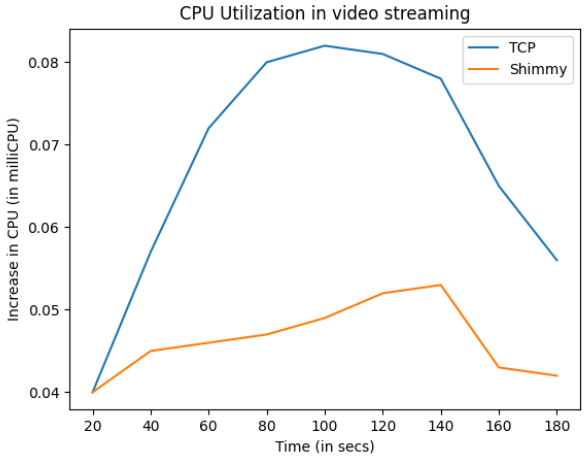

Fig. 7: Comparison of CPU Usage in video streaming between TCP and Shimmy

Figures 7 and 8 show CPU and memory usage during this transfer as observed from the Google Cloud Dashboard for Kubernetes, which uses the GKE usage metering. We note that both the CPU and memory usage is significantly lower for Shimmy compared to TCP. We see a total increase of 0.012 milliCPU and 2 MiB for transferring the stream over Shimmy

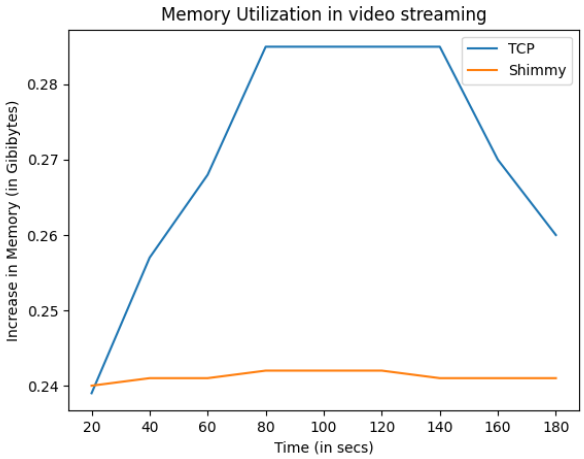

Fig. 8: Comparison of Memory Usage in video streaming between TCP and Shimmy

compared to a total increase of 0.042 milliCPU and 46 MiB for TCP. Further, we note that both CPU and memory usage increase with time during video transfer for both Shimmy and TCP. However, this increase is much lower for Shimmy compared to TCP.

E. RDMA Performance

Figure 9 provide a performance comparison between TCP and RDMA when the two communicating containers are running on two different servers. As we can see, the perfomance gain of RDMA over TCP is significant, up to 41x improvement, which is much higher than the performance improvement when the two containers are running on the same server and using Shimmy. The key reason for this significant performance improvement is that the networking overhead incured in TCP-based communication is quite significant.

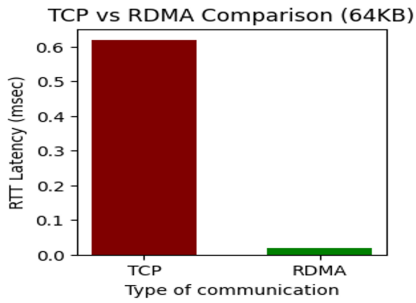

Fig. 9: Latency: RDMA vs TCP

V. Conclusion

Shimmy addresses a fundamental bottleneck in building complex, latency-sensitive, microservices-based applications at the edge, namely the performance of inter-container communication. A significant feature of Shimmy is that it is integrated with Kubernetes. It provides a 3-4x improvement in communication latency over UDP and TCP and a 20x improvement over RabbitMQ while significantly reducing the overall CPU and memory usage. In future, we plan to evaluate Shimmy against with other communication protocols such as QUIC (Quick UDP Internet Connections) or SCTP (Stream Control Transmission Protocol). In addition, since specialized processing units such as accelerators are increasingly being used at the edge, our future goal is to incorporate support for CXL (Compute Express Link) that provides cache-coherent interconnect for processors, memory expansion, and accelerators.

References

[1] Chen, R., Li, S. & Li, Z. From Monolith to Microservices: A Dataflow-Driven Approach. *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. pp. 466-475 (2017)
[2] Kanev, S., Darago, J., Hazelwood, K., Ranganathan, P., Moseley, T., Wei, G. & Brooks, D. Profiling a warehouse-scale computer. *ISCA '15 Proceedings Of The 42nd Annual International Symposium On Computer Architecture*. pp. 158-169 (2014)
[3] Hassan, N., Yau, K. & Wu, C. Edge Computing in 5G: A Review. *IEEE Access*. **7** pp. 127276-127289 (2019)
[4] Wikipedia Remote Direct Memory Access. *Wikipedia*. (2021,1), https://en.wikipedia.org/wiki/Remote_direct_memory_access
[5] Kubernetes What is kubernetes?. *Kubernetes*. (2022,4)
[6] Abbasi, U., Bourhim, E., Dieye, M. & Elbiaze, H. A Performance Comparison of Container Networking Alternatives. *IEEE Network*. **33**, 178-185 (2019)
[7] Kim, D., Yu, T., Liu, H., Zhu, Y., Padhye, J., Raindel, S., Guo, C., Sekar, V. & Seshan, S. FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds. *16th USENIX Symposium On Networked Systems Design And Implementation (NSDI 19)*. pp. 113-126 (2019,2).
[8] Budigiri, G., Baumann, C., Mühlberg, J., Truyen, E. & Joosen, W. Network Policies in Kubernetes: Performance Evaluation and Security Analysis. *European Conference On Networks And Communications*. pp. 1-6 (2021,7), https://lirias.kuleuven.be/3552096
[9] Suo, K., Zhao, Y., Chen, W. & Rao, J. An Analysis and Empirical Study of Container Networks. *IEEE INFOCOM 2018 - IEEE Conference On Computer Communications*. pp. 189-197 (2018)
[10] Abranches, M., Goodarzy, S., Nazari, M., Mishra, S. & Keller, E. Shimmy: Shared Memory Channels for High Performance Inter-Container Communication. *2nd USENIX Workshop On Hot Topics In Edge Computing (HotEdge 19)*. (2019,7).
[11] Hobson, T., Yildiz, O., Nicolae, B., Huang, J. & Peterka, T. shared memory Communication for Containerized Workflows. *2021 IEEE/ACM 21st International Symposium On Cluster, Cloud And Internet Computing (CCGrid)*. pp. 123-132 (2021)
[12] Doxygen Shared Memory Packet Interface (memif) Library. *FD.IO VPP: Shared Memory Packet Interface (memif) Library*. (2017,11), https://docs.fd.io/vpp/17.10/libmemif_doc.html
[13] Bradski, G. The OpenCV Library. *Dr. Dobb's Journal Of Software Tools*. (2000)
[14] RabbitMQ. Available at https://www.rabbitmq.com/.
[15] AMQP. Available at http://www.amqp.org.
[16] Xue, J., Miao, Y., Chen, C., Wu, M., Zhang, L. & Zhou, L. Fast Distributed Deep Learning over RDMA. *Proceedings Of The Fourteenth EuroSys Conference 2019*. (2019).
[17] Fent, P., Renen, A., Kipf, A., Leis, V., Neumann, T. & Kemper, A. Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory. *2020 IEEE 36th International Conference On Data Engineering (ICDE)*. pp. 1477-1488 (2020)
[18] Apache Pulsar. Available at https://pulsar.apache.org/.
[19] Apache Kafka. Available at https://kafka.apache.org/.
[20] Nikhil, A.and Chandar, V. Benchmarking Apache Kafka, Apache Pulsar, and RabbitMQ: Which is the Fastest?. Available at https://www.confluent.io/blog/kafka-fastest-messaging-system/#: :text=Throughput%3A%20Kafka%20provides%20the%20highest,strong%20durability%20and%20high%20availability.