Shimmy: Shared Memory Channels for High Performance Inter-Container Communication

Marcelo Abranches*, Sepideh Goodarzy*, Maziyar Nazari*, Shivakant Mishra, Eric Keller
University of Colorado, Boulder

Abstract

With the increasing need for more reactive services, and the need to process large amounts of IoT data, edge clouds are emerging to enable applications to be run close to the users and/or devices. Following the trend in hyperscale clouds, applications are trending toward a microservices architecture where the application is decomposed into smaller pieces that can each run in its own container and communicate with each other over a network through well defined APIs. This improves the development effort and deployability, but also introduces inefficiencies in communication. In this paper, we rethink the communication model, and introduce the ability to create shared memory channels between containers supporting both a pub/sub model and streaming model. Our approach is not only applicable to the edge clouds but also beneficial in core cloud environments. Local communication is made more efficient, and remote communication is efficiently supported through synchronizing shared memory regions via RDMA.

1 Introduction

Cloud computing has had a tremendous impact on the way applications are built and deployed. While largely centralized in hyperscale data centers, this impact is poised to extend to the edge. This move has already started with the emergence of cloud and co-location providers focused on the edge, such as Vapor.io [14] and EdgeConneX [6]. The reason is clear, in being closer to the users, edge clouds can provide more reactive services, and in being closer to the devices, edge clouds can process the data closer to the source, resulting in a more efficient design.

In the hyperscale clouds, containerization (e.g., with Docker [4] or rkt [11]) is quickly becoming widely used because they provide a lightweight environment which supports isolation of resources. The major public cloud computing providers, including Amazon Web Services [1], Microsoft Azure [10], and Google Cloud Platform [7] have embraced container technology with hosted container services built around container orchestration technology such as Kubernetes [9] or Mesos [2].

With this, application developers have already been rethinking the design of applications, namely toward microservices. This design promotes breaking applications into smaller, independent parts which can communicate through a well defined interface. In doing so, this enables faster development cycles, better failure isolation, and finer grained scalability.

The communication plays a key role in this new design. Application developers prefer to have a low latency platform to make microservice containers to communicate with each other as quickly as possible. Similarly, cloud providers want to utilize their resources as much and as efficiently as possible. Efficiency becomes more important in edge clouds, where resources may be more limited than core clouds.

While developers are embracing new design principles, such as with the 12 factor app [13], the communication channels are largely assumed to need to follow traditional means. In particular, applications continue using Berkeley sockets with TCP/IP, but that isn't really a requirement from the underlying infrastructure. In fact, container orchestration includes a layer (e.g., the CNI [3] layer with technologies such as Weave [15] and Istio [8]) which map between the traditional network interfaces applications use, and the flexible and dynamic (and largely local area) communication the container systems are deployed around. Of course, this layer introduces extra latency and other inefficiencies. Recent work demonstrated the ability to use RDMA to transfer data between containers more efficiently [20], but we ask if there's something more.

In this paper, we propose that we rethink the communication interface of microservices, as we have with the rest of the design of applications. In particular, we propose that we build around a shared memory channel. That is, we treat containers as connected via a collection of shared memory channels that can be either a bi-directional stream (as in TCP/IP) or a publish/subscribe channel. This optimizes local communication (between containers co-located on the same server), but

*All three authors contributed equally to this paper

can also support fast and efficient remote communication by synchronizing memory regions via RDMA technology.

Shared memory as a widely used network channel is impractical as a traditional means being that we traditionally need to support independently operating 'machines', which TCP/IP is ideally suited for. But with modern infrastructures we can assume these applications are run within a container orchestration framework, which provides control over both the communication interface, and the communication medium, which makes this proposal practical.

In the rest of this paper, we provide a motivating example edge computing application which follows the model of multiple, communicating containers (Section 2). We then overview our architecture (Section 3), and how we broker the communication (Section 4) and setup shared memory channels (Section 5). We then discuss our initial prototype (Section 6), which is a first step toward feasibility, and evaluation (Section 7) which demonstrates the lower latency (important for edge computing). We then wrap up with conclusions and future work (Section 8).

2 Motivating Example

In this section we are going to talk about an example in which we will identify the necessity of existence of a high-throughput low-latency communication in a pipeline of containers whether in core clouds or edge gateways. Let's assume we have an image processing application consists of multiple containers, in which an image, in every step, will go through an application running in a container and the generated result will be the input for another container or set of containers.

In Figure 1 we have the image raw data as an input to the first container which will extract metadata and put it in a good shape for processing. The result will be passed as an input to the second container which will process metadata. The third one will be either a container playing role as an object detection algorithm application (Tensorflow) or, for the sake of user privacy, blurring the face of a person which is present in the picture. In the next step, based on the previous step, we will either resize the image provided by the object detection application and output it if the data comes from the object detection application container or run the object detection algorithm on the blurred image at first and then, we will resize the result image and output it. This is sort of tree-structured pipeline of containers in which we need passing data as quickly as possible between them [19].

These types of applications motivated us to rethink the communication model to provide one that is efficient for local and remote communication and support streaming and pub/sub models of data transfer.

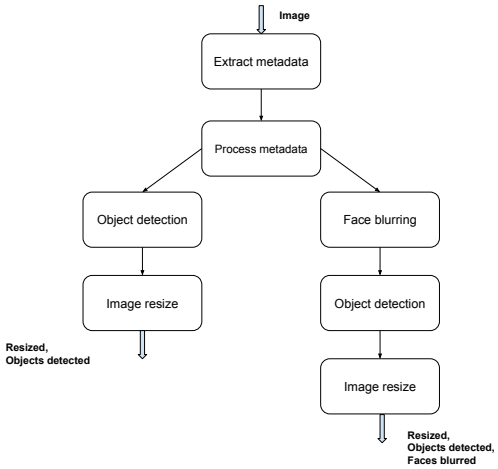

Figure 1: A motivating example showing an image processing pipeline

3 Overview

Shown in Figure 2 is an overall view of how the shared memory channels fit into the overall system. We assume that applications are run in a container orchestration system such as Kubernetes. These systems consist of hosts that can run containers. Each host runs a container management daemon (e.g., a docker daemon) which provides an interface to the orchestrator to run a container on that specific host. Running the infrastructure is the orchestrator which manages what containers should be running, and determining which hosts to run them on (via the scheduler). This functionality is exposed to administrators via an API.

We add two new components to this overall orchestration system, a single instance of Shimmy agent running on each host shared by all containers on the host (Figure 2), and the Shimmy master, which there is logically only one. We introduce the ability for admins to define new shared memory communication channels as part of the overall multi-container application. The Shimmy master is then responsible for tracking the container deployments and interfacing with the Shimmy agents on the hosts in order to ensure that the communication channels that were defined are established between the associated containers (whether local or remote). As future work, we envision the schedules becoming channel aware and be designed to optimize the overall amount of local communication. In the following sections we will detail the Shimmy master and Shimmy agent.

4 Shared Memory Channel Object

Our design is based around the idea that container orchestration frameworks are becoming very much the operating system of the cloud. That is, applications consist of multiple containerized services that interconnect with one another and are deployed and managed by an orchestrator. With this,

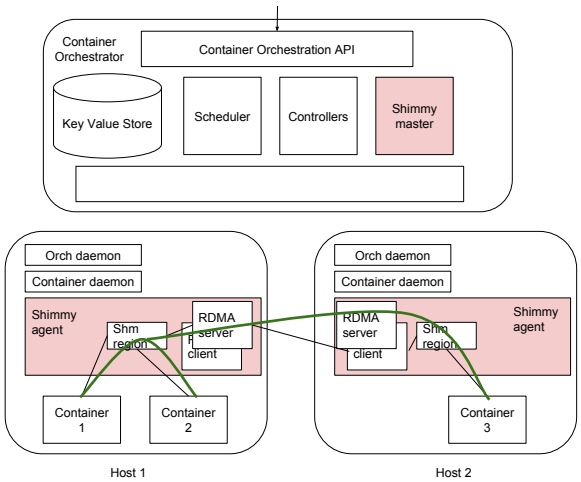

Figure 2: System Architecture (red highlighted components are where Shimmy sits, the green darkened line represents a pub/sub channel with container 1 as publisher and 2 and 3 as subscribers)

we envision that rather than applications dictating their communication patterns (e.g., by explicitly specifying endpoints within the application code), they are described through the configuration system which describes the overall application.

To explain, consider how one aspect of networking in Kubernetes works today. A Service is a resource type which provides a single entry point to a group of Pods (containers) running the same application. To create a Service, you define a yaml file which specifies, among other things, the network information (port numbers). This yaml file also specify a selector (e.g., app:myappname, where app is the key and myappname is the value), which is used to match against Pods. Any Pod that species that key-value pair as a tag becomes associated with that service (meaning, it is considered in the set of Pods which the Service provides an entry to). Istio [8] extends this concept to configure proxies running in side cars to the containers that can apply access control rules to the traffic. In each case, it was outside of the application code where aspects of the connectivity were defined.

Likewise, we envision a new resource type, SharedMemoryChannel, which can be described through a yaml file which would include (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 SharedMemoryChannel object, the Shimmy master would note this and then notify the Shimmy agent on the host for which the Pod was deployed, and then subsequently attach a shared memory channel of the desired type to the Pod. We detail the actions of the agent in the next section.

5 Establishing Shared Memory Channels

As previously mentioned, the Shimmy agent runs on each server. Much like the kubelet and docker agent, it provides the functionality necessary to perform some action on a given host. It has a well defined API that the Shimmy master uses to indicate what it wants done (since it has the cluster-wide perspective, and manages the application as a whole). Here, we outline how shared memory channels are established.

Allocating memory: The Shimmy agent is a process that runs on the host. When a channel needs to be established on the host it is resident on, it will allocate memory to be shared. This is done through the shmget() system call. This call returns a shm_id which identifies the shared memory region. For pub/sub type communication, we only need one shared memory region per host (even if there are multiple subscribers) that has a container attached to the channel. For streaming type communication, we need to set up one shared memory region per direction on each host which has a container attached to the channel.

Attaching to a container: The shared memory channel is between the process in a container and the Shimmy agent. To make this happen, we need to break the barrier of the isolation that containerization provides. Fortunately this is simple, and just requires specifying that the inter-process communication namespace can be shared between the container and the host (done with –ipc="host" in Docker, or hostIPC=true in Kubernetes). Then, the container just needs to know the shm_id to use, which, following the principles of the 12 factor apps, could be passed in as an environment variable.

Application interface: Internal to the application, the application needs to initialize its end of the channel, much as it would need to setup a network socket even with the network interface already attached. The shm_id is needed to pass to the operating system to indicate the shared memory region to use. Once done, then it is simple copying into and out of memory. The semantics of the memory region need to be consistent, so an API is used to provide an ability to read and write to the channel.

Synchronizing across machines: Shared memory is an inter-process communication mechanism, and therefore is highly efficient. In a full microservice model, some containers that need to communicate will inevitably end up on different hosts. To do this, we leverage RDMA technology. RDMA enables directly copying from one region of memory from one host to another without involving the processor. This requires the Shimmy agent to serve as both a RDMA server and an RDMA client. For channels involving a remote container, the agent will continuously transfer data using RDMA.

6 Prototype

As a first step, we focused on creating the Shimmy agent, and supporting the publish/subscribe communication type. Although we have mentioned we support streaming communication type, pub/sub model is a common type of communication in the edge platforms and it is comparable to the well-known systems like Apache Kafka and Mosquitto. This allows us

to demonstrate setting up shared memory based communication channels between containers, supporting both local and remote communication. We have not fully integrated into Kubernetes, so, currently the application needs to do the work to create the shared memory objects (rather than being described in a Kubernetes configuration, which then creates the communication channels transparently to applications running in individual containers).

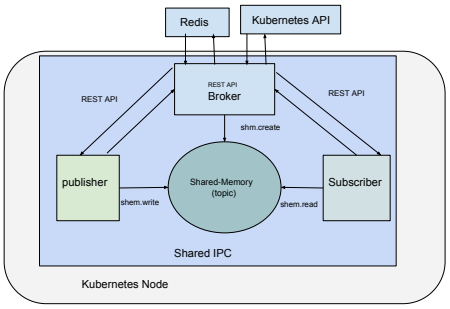

Figure 3: Prototype design

Figure 3 shows the current architecture of our proposal. The Broker (the Shimmy agent) is a python application that is responsible for managing the creation and subscriptions of topics. We are using the *sysv_ipc* library to perform tasks related to shared memory. In order to allow publishers, subscribers, and the broker to attach to common shared memory, we are using the Kubernetes option *hostIPC: True*, to that all of them share the same IPC namespace. The broker interacts with Kubernetes API, to determine the location of containers.

The publishers and subscribers interact with the broker using REST APIs for setting up topics and subscriptions, and other management tasks. Data communication is done through shared memory.

After setting up topics and subscriptions, applications use the shared memory channels to perform the data transfer, using the *sysv_ipc* libraries. If the publisher and subscriber are co-located, they will both be accessing the same shared memory region. If they are not, an RDMA client or server will synchronize the memory regions between the two servers.

7 Evaluation

In the preliminary evaluation of our system we have compared Shimmy with the Mosquitto message broker [16] and with Kafka [17] as they are both well known data streaming platforms that also use the publisher/subscriber model. We have built docker containers for Mosquitto, Kafka, and also for the publishers and for the subscribers in each of this environments. Mosquitto implements the MQTT protocol [18] and uses the TCP/IP stack for communication. Kafka also uses TCP/IP.

We compared the systems in two scenarios: local communication where publishers and subscribers were on the same host, and remote communication, where publishers and subscribers were on different hosts. For remote communication Shimmy was using the shared memory with the RDMA synchronization mechanism described in the previous sections.

7.1 Setup

We performed our tests using two Cloudlab Servers (1x Xeon E5-2450 processor (8 cores, 2.1Ghz), 16GB Memory (4 x 2GB RDIMMs, 1.6Ghz), 1 x Mellanox MX354A Dual port FDR CX3 adapter w/1 x QSA adapter) running Ubuntu 16.04. For our system we have built docker containers for the broker, publisher, and subscriber.

The tests consisted in publishing 16 Bytes and also 100 KB messages. We measured average latency and throughput for each system to make the subscriber receive ten thousand, a hundred thousand and a million of messages. We do not include the setup times (e.g topic subscriptions) in our measurements for any of the systems.

7.2 Results

In Figures 4 and 5 we can see the average throughput and latency for each system with co-located (local) publisher and subscribers. The Shimmy Broker had better throughput and lower latency than Mosquitto and Kafka in all the tests that we performed due to being able to communicate via shared memory rather than going through the kernel networking stack. As the size of messages increases, the overhead to read and write data in the memory channels also increases, but even for 100 KB messages our system had better performance than Kafka and Mosquitto.

Figure 6 shows the average throughput and latency for 16 Bytes messages when the publisher and subscriber are not co-located, and therefore have to communicate over a network. We can see that for remote communication, Shimmy also had better average throughput and lower latency than Mosquitto and Kafka.

Of course, we need to more fully evaluate the system, testing the scalability and robustness, but initial results demonstrate the potential of this approach. It should be noted that, at this point the synchronization of the shared memory channel is not complete as a result of which we were not able to report reliable results for bigger message size (100KB) in remote communication.

8 Conclusions and Future Work

As applications are moving toward the edge of the network, closer to the end users and devices, latency is becoming a critical factor. In this paper we proposed a re-thinking of the communication model as we evolve the application development and deployment model toward microservices. We propose a model around shared memory channels that lends

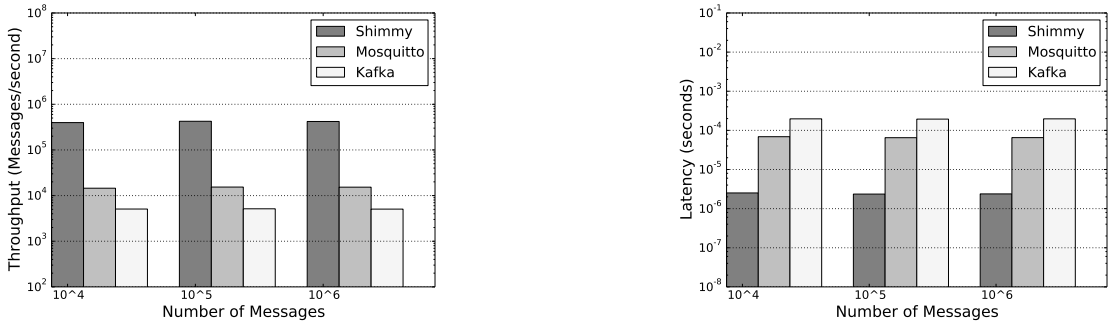

Figure 4: Throughput and latency of local communication for 16 bytes messages (Y-axis is in log scale).

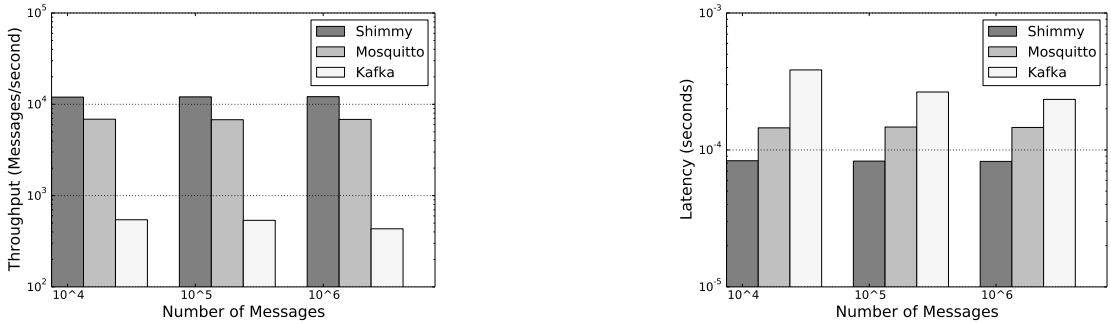

Figure 5: Throughput and latency of local communication for 100KB messages (Y-axis is in log scale).

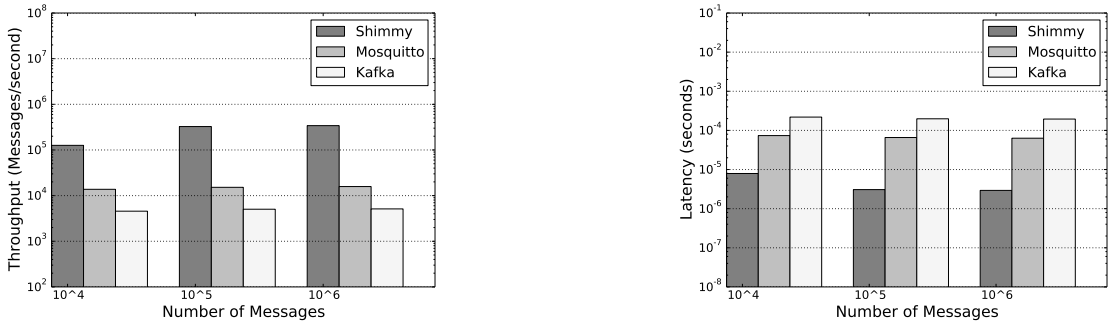

Figure 6: Throughput and latency of remote communication for 16 bytes messages (Y-axis is in log scale).

itself to efficient local communication. To enable remote communication, we extend by synchronizing memory through RDMA.

Our work is only a first step. There's a great deal of work to fully realize and evaluate this new approach. Specifically, we intend to focus on integration into Kubernetes, first creating a SharedMemoryChannel object, and then enabling the scheduler to be application aware – meaning, it will attempt to optimize the amount of local communication by scheduling containers that communicate with each other to run on the same host.

As Shimmy will be running on Edge/Cloud environments (possibly in multi-tenant scenarios), we will provide mechanism for isolating the shared memory channels. Currently, we are integrating our system with Kubernetes policies and SELINUX [12], so that we can control which Pods from which namespaces can access each shared memory channel.

Besides, Shimmy agent should be able to handle concurrent data consumption by multiple instances of an application. Let us say we have 2 instances of a microservice component. In pub/sub communication model, clearly both subscriber instances have access to all of the data available in the shared memory channel, whereas in streaming communication model Shimmy agent will take care of concurrent access to the data and the data consumed by an instance will not be available to the other instance. Thus, they can do their work independently and parallelly. We can think of it as a load balancing effort done by the system.

Discussion

(a) what kind of feedback they are looking to receive: The community can help us with feedback regarding how can we improve our communication models and infrastructure to provide a complete low latency/high throughput platform for edge/cloud computing. What critical functionality in missing in our current proposal? What kind of evaluation we could perform in order to show the value of our platform? What are other platforms that we should compare our platform with? Are there other technologies that we could leverage to improve our proposal? Are there other communication models or paradigms (other than pub/sub and streaming) that we should provide?

(b) the controversial points of the paper: In this paper we are re-thinking how inter-container communication should be done in a edge/cloud computing environment. Traditionally, inter container communication occurs through IP overlay networks. We propose a high-performance platform for container communication using shared memory, on a cloud native platform for edge clouds. We are building our platform on the top of Kubernetes. Our model does assume that we will move toward describing the communication patterns of an application (where an application consists of multiple, inter-communicating micro-services each running in containers) at the orchestration layer (as configuration) rather than within applications.

(c) the type of discussion this paper is likely to generate in a workshop format: We see a trend towards the use of kernel bypass technologies to improve network performance (e.g. DPDK [5] and Netmap [21]). We follow this trend and propose a lightweight mechanism for inter container communication that bypasses the kernel networking stack. This paper should generate a discussion about new methods for providing kernel bypass communication systems, that leverages mechanisms other than TCP/IP kernel stack bypass.

(d) the open issues the paper does not address: In the future work session, we described the issues that we are currently not addressing, but are in our road map. In general, it's an early prototype that focused on the pub/sub system and largely leaves the full Kubernetes integration implementation as future work. Also, applications would need to be modified to take advantage of Shimmy's architecture, but we hope that the performance benefits will make it worth.

(e) under what circumstances the whole idea might fall apart: As we leverage container orchestrators to build our platform, our proposal may fall apart if the interest, adoption and development of this kind of platform starts to slow down (which we do not see this happening in a near future). Also currently we leverage specialized hardware (RDMA NICs) to perform remote IPC, so the availability of this kind of hardware is critical to our proposal.

Acknowledgement

This work was supported in part by NSF Grants 1652698 (CAREER) and 1406192 (SaTC) and the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

References

[1] Amazon Web Service. https://aws.amazon.com.

[2] Apache Mesos. http://mesos.apache.org.

[3] Container Networking Interface. https://github.com/containernetworking/cni.

[4] Docker. https://www.docker.com.

[5] DPDK, Data Plane Development Kit. https://www.dpdk.org/.

[6] EdgeConneX. https://www.edgeconnex.com/.

[7] Google Cloud. https://cloud.google.com.

[8] Istio. https://istio.io/.

[9] Kubernetes. https://kubernetes.io.

[10] Microsoft Azure. https://azure.microsoft.com/en-us.

[11] RKT. https://github.com/rkt/rkt.

[12] SELINUX. https://selinuxproject.org/page/Main_Page.

[13] The twelve factor app. https://12factor.net/.

[14] Vapor.io.

[15] Weave. https://www.weave.works.

[16] Eclipse Mosquitto: An open source MQTT broker. https://mosquitto.org/, 2018.

[17] Kafka - A distributed Streaming Platform. http://kafka.apache.org/, 2018.

[18] MQ Telemetry Transport. http://mqtt.org/, 2018.

[19] Akkus Istemi Ekin et al. Sand: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, 2018. USENIX Association.

[20] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. Freeflow: Software-based virtual RDMA networking for containerized clouds. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 113–126, Boston, MA, 2019. USENIX Association.

[21] Luigi Rizzo. netmap: A novel framework for fast packet i/o. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 101–112, Boston, MA, 2012. USENIX Association.