ConMon: an Automated Container Based Network Performance Monitoring System

Farnaz Moradi, Christofer Flinta, Andreas Johnsson, Catalin Meirosu
Cloud Technologies, Ericsson Research, Sweden
Email: {farnaz.moradi,christofer.flinta,andreas.a.johnsson,catalin.meirosu}@ericsson.com

***Abstract*—The popularity of container technologies and their widespread usage for building microservices demands solutions dedicated for efficient monitoring of containers and their interactions. In this paper we present ConMon, an automated system for monitoring the network performance of container-based applications. It automatically identifies newly instantiated application containers and observes passively their traffic. Based on these observations, it configures and executes monitoring functions inside adjacent monitoring containers. The system adapts the monitoring containers to changes driven by either the application or the execution platform. The evaluation results validate the feasibility of the ConMon approach and illustrate its scalability in terms of low overhead on compute resources, moderate impact on applications, and negligible impact on the background network traffic.**

I. Introduction

Containerization technologies are becoming more prevalent in development and operational environments. Containers are considered as lightweight alternatives to virtual machines [1], since they have low overhead and can be created and stopped quickly [2]. Each container typically runs a single process and can communicate with other containers using virtual networks. Containers are major enablers for microservice architectures, where applications are distributed among a number of instances, each running in an isolated container. These instances can be dynamically started, stopped, and changed. The dynamic nature of microservices makes monitoring them a challenging task.

Tools for monitoring containers typically provide compute resource utilization metrics such as CPU, memory, and block I/O usage as well as counters for packets and bytes transferred over container network interfaces. However, none of the existing tools provide an automated solution for dynamic monitoring of network performance metrics such as delay, jitter, and packet loss between application containers.

Another popular approach involves injecting monitoring code into application containers. This requires preparing special images of the application containers that enable executing additional code within the container envelope, or running extra processes inside the container after it has been started. However practical, such method contradicts the best practices of microservice architectures where each container runs a single isolated process that can be easily restarted or moved with little dependence on other components of the software-defined infrastructure. It increases the risk that issues with the monitoring process (such as crashes or overuse of resources) would affect directly the application by limiting the performance or, in extreme cases, making it crash. Furthermore, the life-cycle management of the monitoring functionality becomes more complex due to both potential effects on the application performance and limitations regarding IP addressing for individual containers.

In microservice architectures, application component instances deployed within containers communicate only within dynamically changing subsets of instances. The communication pattern depends for example on where an initial request is assigned for handling and where the data associated to that request might be stored. Clearly, a full mesh monitoring of all the possible communication paths for several network metrics simultaneously would introduce a significant messaging overhead, in particular when it would be carried at sub-second intervals that ensure fast failure detection for mission-critical applications. It would thus be desirable to develop solutions that are capable of reducing the unnecessary cost of monitoring links between the containers that do not communicate with each other.

To address these challenges we propose ConMon, a distributed and automated solution for observing network metrics in container execution environments. ConMon eliminates the manual selection and configuration of measurement points. It keeps up with the dynamic nature of containerized applications and adapts the monitoring to continuously perform accurate, both in terms of location and metric precision, and timely performance measurements. In ConMon, the containers that communicate with each other are automatically discovered and network performance is monitored only between them. The automatic discovery and setup of monitoring containers in ConMon provides uninterrupted monitoring in case of container scaling and migration without manual intervention.

In this paper ConMon and its different components are presented and evaluated by investigating its feasibility, scalability, and in particular the effects of passive traffic observation on the performance of the application containers and on the system resources. The remainder of this paper is organized as follows. Section II presents the related work. Section III describes ConMon and its components. In Section V the testbed and the performance evaluation results are presented. Section VI presents some discussions and finally Section VII concludes the paper.

978-3-901882-89-0 @2017 IFIP

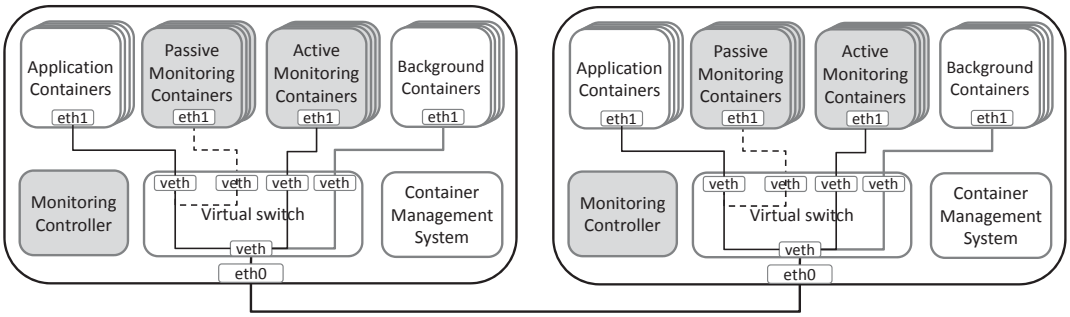

Fig. 1. ConMon components in our testbed.

II. Related Work

A wide variety of monitoring solutions developed for cloud environments exist both commercially and open sourced. These solutions range from generic tools that are extended for monitoring in cloud environments such as Nagios [3] to cloud-specific ones such as Amazon CloudWatch [4] for monitoring AWS cloud resources and applications, and Ceilometer and Monasca [5] for monitoring in OpenStack environments [6]. The requirements and properties of monitoring solutions for cloud and a survey of existing platforms can be found in [7].

Some existing cloud monitoring systems have added functionality to support container monitoring and some new solutions have been designed and created specifically for container environments. Most of the existing tools gather resource usage metrics such as CPU, memory, and block I/O usage for containers running on a host machine. Network metrics are limited to the number of packets and bytes observed on an interface. For example, Docker [8] provides a stats API which allows access to a live stream of counters for running Docker containers. Tools such as CAdvisor [9] create a monitoring container to monitor resource usage and network interface counters of containers on a host.

Other container monitoring systems such as Dynatrace [10] allow monitoring the processes and applications running inside the containers. The monitoring agent on each host automatically detects when a container is started and injects the desired monitoring function into the container. Such solutions are designed to monitor the application functionality within containers and not the network performance between the containers. Another container native monitoring solution is Sysdig [11] which installs a Linux kernel module to observe the system calls from the containers and other Operating System events and allows zero-configuration monitoring from a single monitoring container. These tools do not support autonomous and dynamic monitoring of the network performance between microservice containers.

Network monitoring in cloud environments can also be performed by general purpose monitoring systems. An example is sFlow [12], which has added support for containers to obtain standard sFlow performance metrics, but does not provide end-to-end measurements by itself. In [13] Pingmesh is introduced to ensure continuous network monitoring with maximum measurement coverage for troubleshooting network performance degradations across a datacenter or between datacenters, using active measurements. However, it is not designed for monitoring network performance perceived by applications running inside virtual machines or containers. Network monitoring in Software Defined Networks (SDN) have also received considerable attention. In these environments network monitoring is typically performed using OpenFlow messages in OpenFlow-enabled virtual and physical switches in the network. Examples of such methods are latency [14], link utilization [15] and low-overhead packet loss measurements [16].

The effects of packet mirroring, which is required for passive network measurements, on network traffic have been studied before. It was shown in [17] that port mirroring in commodity switches can slightly increase packet drops for non-mirrored traffic due to shared buffer space. In [18] it was shown that data collection on hypervisor virtual switches does not impact the user traffic because the capacity of the virtual switches is higher than the 10 Gbps capacity of the NIC card. However, dumping traffic adds extra cost to memory throughput. In this paper we further study the impact of passive monitoring in a container execution environment.

III. The ConMon Monitoring System

The architecture of ConMon is shown in Figure 1. ConMon consists of three main components: Monitoring Controllers (MCs), Passive network Monitoring (PM) containers, and Active network Monitoring (AM) containers. Different types of Monitoring Functions (MFs) can be executed inside the PM/AM containers. ConMon relies on the underlying Container Management System to obtain container life-cycle events and instantiate or remove monitoring containers. It also uses virtual switches for tapping the application traffic.

A. Monitoring Controller

The core components of the ConMon system are the Monitoring Controllers (MCs) which run on each physical server and communicate with each other in a distributed fashion. The communication can be implemented in several ways such as via a distributed database, a messaging system, or a central management system.

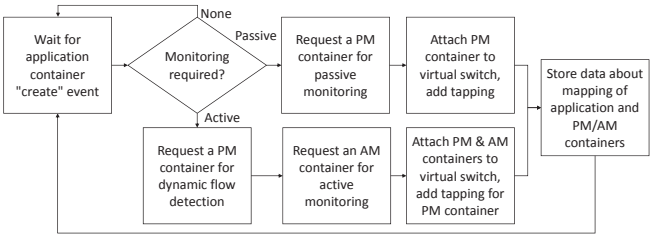

Fig. 2. Steps for monitoring container instantiation by MC.

B. Monitoring Containers

The monitoring functions (MFs) are deployed inside monitoring containers, adjacent to the application containers and interconnected via virtual switches (see Figure 1). ConMon allows a variety of different types of active and passive MFs to be deployed in the monitoring containers in order to monitor different network performance metrics such as packet loss, delay, jitter, and available path capacity.

Passive Monitoring (PM) container: A PM container receives a copy of packets from the monitored application on its interface on the virtual switch either via port mirroring or tapping. The MF which runs inside the PM container, e.g., *tcpdump*, can capture, filter and timestamp the copy of packets. Additionally, an MF can estimate different performance metrics using different algorithms such as COLATE [19] for measuring latency between containers [20].

The MF inside a PM container can also be used for identifying changes in the application communications, e.g., a new flow or an expired flow. This information can be used by MC to dynamically (re-)configure active or passive MFs in reaction to the changes. A single PM container can be used for monitoring multiple application containers if they belong to the same entity, e.g., tenant.

Active Monitoring (AM) container: An AM container is connected to the same virtual switch as the application container. An active MF, a.k.a. probe, runs inside the AM container, and injects probe packets in the network which are received in another probe. The probe packets can be collected in the receiver probe or reflected back to the sender probe. Some measurements require the probe packets to be timestamped in both probes at sending and arrival events. In this way one could study interaction between probe packets and the cross traffic and draw conclusions about the network characteristics and the cross traffic dynamics. Examples of active MFs that can run inside AM containers include ICMP ping, TWAMP [21], *iperf* [22], and *netperf* [23]. An AM container can be shared by multiple application containers that reside on the same physical server.

C. Automatic instantiation of monitoring containers

Each PM or AM container is instantiated and configured by a local MC which resides on the same physical server. The steps taken by MC to setup monitoring containers are shown in Figure 2. MCs continuously listen to events generated by a local or remote container management system or orchestrator, e.g., Docker [8]. Once an event regarding instantiation of an application container is observed, the MC requests from the container management system to start the required monitoring containers. MC then attaches the created monitoring container(s) to the virtual switch to which the application container is connected. For passive network monitoring, MC also configures the virtual switch to tap/mirror the application traffic to the PM container. For active network monitoring, in order to adapt the active monitoring sessions to the application communications both a PM and an AM container are required which can be instantiated at the same time.

Once an AM or PM container is started and configured, MC has to store the information about mapping of the application container and corresponding monitoring containers. This information, which is required for monitor discovery, can be kept locally or stored in a distributed or centralized database.

D. Automatic discovery of remote monitor(s)

Some metrics, such as latency and throughput, require that the MFs on the sender and receiver side are able to identify each other and exchange messages, such as synchronization information, or probe packets. The information about remote MFs can be provided in advance (e.g., as part of the monitoring intents) or be obtained by the MFs (e.g., by communicating with each other via an overlay network). Otherwise, MCs can provide monitor discovery services to the MFs.

The local MC receives information about the application container flows via the passive monitoring function, e.g., source and destination IP addresses. For each new flow, the MC identifies the monitoring container associated with the corresponding remote container by either directly contacting other MCs or by accessing a centralized/distributed database. If a remote monitoring container exists, MC will receive information about its address and can use it to configure the MF. Otherwise, the local MC can request the remote MC to instantiate and configure a monitoring container so that network performance measurements can be performed. Figure 3 shows a sequence diagram for instantiating active network monitoring between containers on a sender and a receiver host. The figure also shows that once the communication between the application containers is stopped, the MC is notified and can decide to stop or modify the active monitoring session and inform the receiver side MC.

In active network monitoring, if the performance between containers on two servers is being monitored by already existing AM containers, the MCs will not instantiate new AM containers and monitoring sessions. Therefore, the unnecessary cost of exchanging extra probe packets is eliminated.

IV. ConMon Use cases

A. Creating a traffic matrix

ConMon can be used for obtaining a real-time traffic matrix describing the container communications. Information about active flows, including metrics such as latency and packet loss, observed by PM containers are stored locally on the server. A

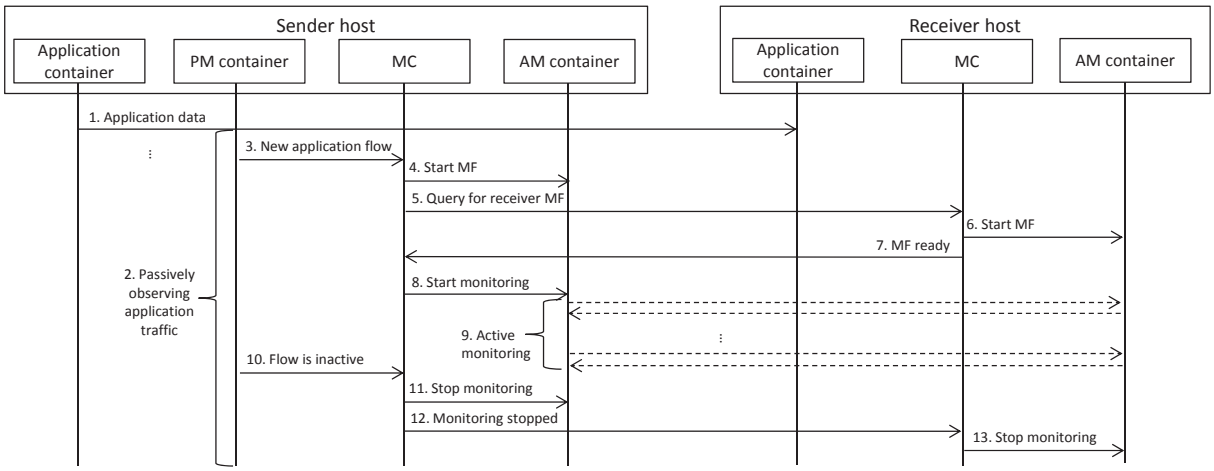

Fig. 3. Sequence diagram of example interactions between different ConMon components for performing adaptive active network monitoring.

snapshot of the entire system is then obtained by collecting these local traffic matrices from different MCs, e.g., using a simple RESTful API.

Traffic matrices can be used for making placement decisions, e.g., to improve the performance of a set of microservices, or reduce the communication costs. Load balancing and rerouting decisions are other examples.

B. Measuring container packet processing time

ConMon can also be used for measuring the packet processing time of a single container. Some network functions such as Firewalls and Deep Packet Inspectors can run in containers as Virtual Network Functions (VNF). Measuring the time it takes for network traffic to be processed by these functions is important for troubleshooting performance degradations in the network. To monitor the processing time, an MF inside a PM container has to record the time when a packet has arrived at container interface together with a signature for the packet. When the corresponding packet leaves the container, the MF identifies the packet using a signature, e.g., based on the IP packet, and records another timestamp and calculates the time difference between the arrival and departure of the packet from the VNF container to measure the packet processing time. A similar approach has been used for measuring metrics such as per-hop delay for VNFs running in an OpenStack environment [24]. The measured data can be used for troubleshooting as well as triggering elasticity actions such as scaling up or out the VNF instance.

C. Network performance measurements between containers

ConMon supports network performance monitoring (e.g., latency) between containers using both passive and active monitoring methods. In passive measurement methods, packet hashes together with timestamp information are stored in specifically defined datastructures at both sender and receiver side PMs. These datastructures can be periodically accessed or exchanged to estimate one-way latency. Examples of such methods include Lossy Difference Aggregator (LDA) method [25] which calculates aggregate latency values, and COLATE method [19] which provides lightweight per-flow latency measurements.

Active network measurements are enabled through ConMon using the AM containers. Examples of MFs to run inside includes ICMP ping, TWAMP, *traceroute*, *iperf*, and *netperf*.

The metrics that are measured either passively or actively can be stored for online or offline analysis, for example for anomaly detection, change detection, and prediction purposes. The measurements can also be used for triggering actions such as migration and scaling of the application containers.

V. Evaluation

In this section we present our testbed and experimental evaluation results. The main focus is on evaluating the setup time for monitoring, the impact of passive monitoring on the host resources and the application containers.

A. Testbed

Figure 1 shows the testbed used for evaluating ConMon. The testbed consists of two physical servers that are directly connected with a 10 Gbps link[1]. Each physical server has 2 Intel Xeon X5660 2.8GHz with 24 CPU cores in total, 48 GB RAM, running Ubuntu 14.04 LTS. In the testbed, we have used Docker (v. 1.10.2) as the container management system for creating application, monitoring, and background containers. These containers are connected to Open vSwitch (v. 2.0.2) virtual switches (OVS). Each virtual switch is responsible for copying the application packets to the monitoring ports, in order to allow a passive MF, within the PM container, to capture and analyze packets. The background containers are used for for evaluating the impact of the ConMon system on other applications.

In all experiments, a sender application container sends traffic to a receiver application container. These containers

[1]In order to study the monitoring impact on the applications we used a direct link to minimize the effects from physical switching/routing and reduce the physical latency between the hosts to a minimum.

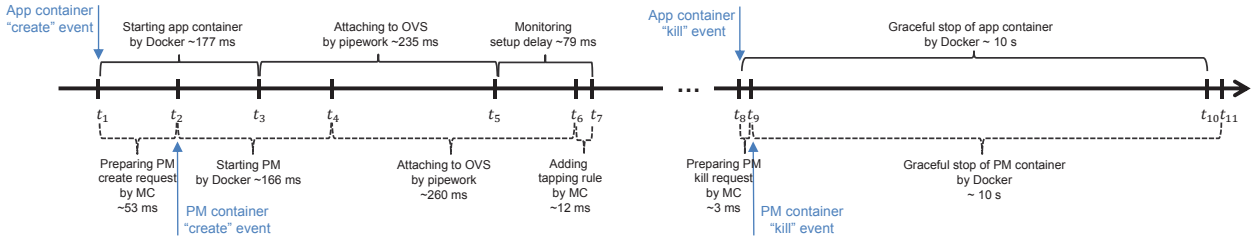

Fig. 4. Setup and teardown time for monitoring.

can reside on the same physical host or on two different hosts. Inside the application containers, the *netperf* tool was used to generate TCP or UDP traffic. Although *netperf* is an active network monitoring tool, in our experiments has been used as a simple representation of application traffic while allowing us to measure the throughput and latency from the application's perspective. The *tcpdump* tool was used for traffic capturing either inside the application containers or inside the PM containers. The tapping in the OVS was performed by adding OpenFlow rules to perform two actions on each packet of the application containers, i.e., forward as normal and send to the monitor port to which the PM container is attached.

Some notes regarding the figures in the next subsections:

- **Base**: refers to experiments with no monitoring.
- **Internal**: refers to monitoring from inside the application containers (used as a representative of solutions requiring monitoring code inside application containers).
- **Adjacent**: refers to monitoring from inside the adjacent monitoring containers.

Each experiment was performed 10 times and average results are presented in the figures.

B. Setup time

In this section we evaluate the performance of the ConMon system by measuring the time it takes for the PM and AM containers to be started and configured by the MCs.

In our testbed, each container, after being started by Docker, is connected to an OVS virtual switch using the *pipework* tool [26] and then a process is executed inside it by Docker. Figure 4 shows the time it takes for each of these steps to be executed in our testbed. Starting a container and attaching it to the switch takes around half a second ($t_5 - t_1$ for application and $t_6 - t_2$ for monitoring container(s)). The figure also shows that in average it takes around 53 ms for MC to prepare a request for instantiating a PM container after detecting the "create" event from Docker ($t_2 - t_1$). The time it takes for instantiating PM container depends on where the container image is stored (locally in our testbed). MC then has to configure the tapping on the virtual switch which in our implementation is done by adding an OpenFlow rule which takes around 12 ms ($t_7 - t_6$). Overall, the delay for setting up monitoring in our testbed has been around 79 ms ($t_7 - t_5$).

Depending on when the application starts to send or receive traffic, the MF can miss some initial application packets during this delay. We tested this by measuring the number of packets which were not observed by *tcpdump* in the PM container. We used an application container which immediately after instantiation and connection to the OVS started to send packets with a rate of 1000 packets per second. We observed that *tcpdump* inside the PM container in average missed the first 76 packets due to the setup time which is consistent with the delay time ($t_7 - t_5$).

The monitoring setup time can be dramatically reduced by starting a PM container and attaching it to the virtual switch in advance on each server, and adding the tapping rule after the application container is attached to OVS. In this case, the monitoring setup time in our testbed is reduced to 28 ms.

An AM container can also be created at the same time as the PM container if active monitoring is also required. However, if remote monitor discovery is required, additional setup time is introduced. For automatic discovery of remote monitoring containers, an MC has to obtain the address of the receiver AM container. In our implementation, each MC after creating a monitoring container updates Consul [27], which is a distributed key/value storage, by adding a record showing the mapping between the address of the application and the monitoring container associated to it. Any other type of centralized or distributed database can be used for this purpose. For monitor discovery, an MC has to query Consul for the mapping. The query time in our testbed when a consul container resides locally is around 3-4 ms even when the storage has more than 800K key-value pairs stored in it. In a realistic implementation the query time depends on the location and the type of the database.

If a remote monitoring container is not running, the local MC can request the remote MC to instantiate and configure one, e.g., an AM container to reflect probe packets. In this case the monitor setup will take longer time including the time for sending the request to the remote MC and starting a monitoring container, i.e., around half a second. The time it takes for a request to be received by an MC depends on the latency between the servers. In our testbed the RTT between containers on two servers has been around 0.15 ms. RTT measurements in an operational datacenter have shown that the median RTT between servers are 0.22 ms and 1.26 ms for intra-racks and inter-rack communications, respectively [13].

Finally, we have measured the time it takes for ConMon to remove a monitoring container after an application container

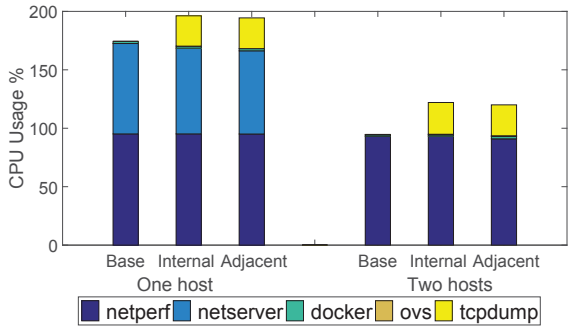

Fig. 5. Total CPU usage (user + kernel) of different processes.

has been removed. In our implementation, once MC detects that an application container is stopped (i.e., detect a "kill" event by Docker), it reacts by sending a request for stopping the corresponding monitoring container(s). This reaction time in our testbed is around 3 ms ($t_9 - t_8$). The graceful stopping and removing the containers by Docker then takes more than 10 s ($t_{10} - t_8$ for application container and $t_{11} - t_9$ for monitoring container).

C. Impact on resource usage

In this section we present the experimental results for the impact of the ConMon monitoring system, in particular passive traffic collection on the resource usage of the servers.

Figure 5 shows the accumulated CPU usage (for both user space and kernel space) for different processes during a *netperf* UDP throughput test between two application containers, with the maximum message size. The CPU statistics are obtained from System Activity Report (SAR) [28], which is a widely-used open source Linux tool. In the figure, the first three bars from the left, show the accumulated CPU usage where the sender and receiver application containers reside on the same server. The three bars on the right show the CPU usage on the sender side server where the sender and receiver containers were placed on two different servers. It can be seen that *netperf* and *netserver* processes use the highest portion of CPU. The *tcpdump* tool which is used for capturing traffic uses the same amount of CPU regardless of where the monitoring is performed, i.e., inside the sender application container (Internal) or inside a PM container (Adjacent). Moreover, Docker and OVS virtual switch have very low CPU usage in all the experiments. The low CPU usage of OVS in this experiment (which is due to the fact that there is only one forwarding rule in the OVS and it is not under load) does not increase by adding tapping to OVS which means that tapping adds negligible extra cost to the cost of data collection.

D. Impact on application performance

In this section we evaluate the impact of the ConMon system on the performance of the applications.

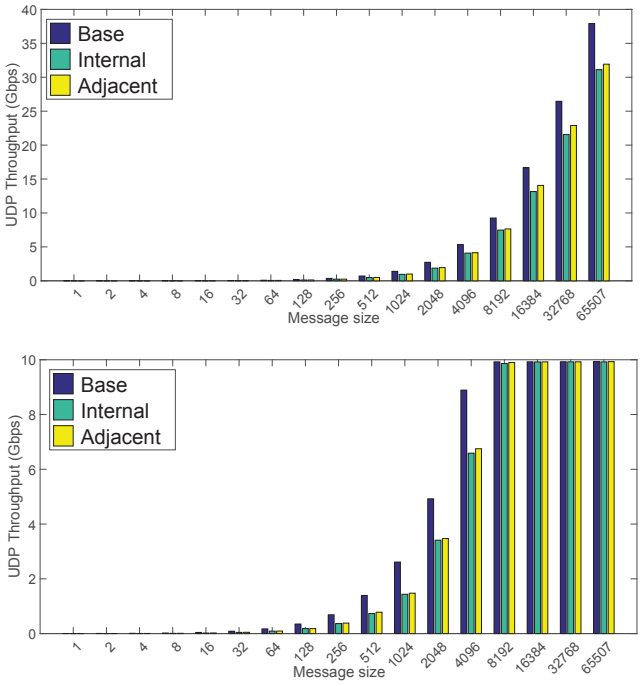

Fig. 6. Throughput measured using UDP traffic between two application containers on (top) the same server and (bottom) two different servers.

Throughput: We have evaluated the impact of passive monitoring on the network throughput of the applications. Figure 6 shows the results for measurements on one server and between two servers obtained from *netperf* tool running inside application containers with different message sizes using UDP packets. It can be seen that a higher message size gives higher throughput by enabling a higher send rate.

In measurements between two servers, where the maximum throughput of 10 Gbps is reached, the passive monitoring does not affect the throughput. For all send rates lower than the link capacity we can see a penalty for monitoring which is translated into a lower throughput. This can be seen in both scenarios, where throughput values for Internal and Adjacent are lower than Base. However, when the send rate is much larger than the link capacity, the link becomes the bottleneck that restricts the throughput. In that case the penalty on the send rates created by monitoring is not seen, since the send rates with monitoring are still over the bottleneck rate.

In the measurements where the sender and receiver application containers reside on the same server, the throughput between them is limited by the capacity of the virtual switch (38.5 Gbps measured peak rate). Measurements with TCP traffic showed similar results and are therefore not presented.

Latency: We investigated the latency between application containers and the effects of monitoring on their perceived delay. Figure 7 shows the Round Trip Time (RTT) measurement results obtained from *netperf*. It can be seen that Internal data collection has increased the RTT by 4.5 μs when the containers reside on the same server and by more than

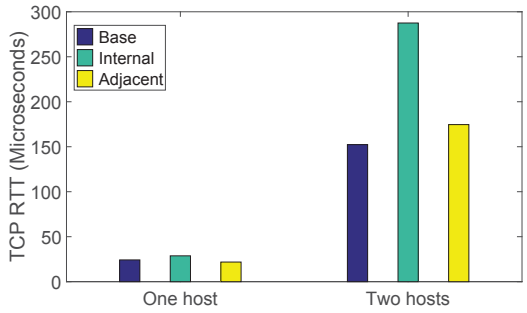

Fig. 7. RTT between two containers on one server and on two servers.

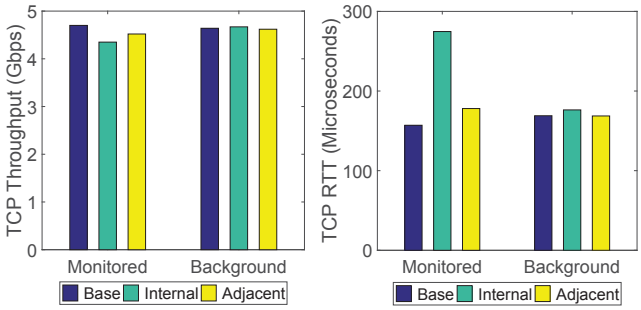

Fig. 8. Impact of monitoring application containers on the background traffic.

130 μs when the containers are located in two servers. Data collection in adjacent monitoring containers has less impact on the application RTT which was negligible when the containers reside on the same server and increased only by 22 μs when the containers reside on two servers.

Packet loss: In our evaluations we have not observed any packet loss according to the counters reported by the virtual switches. However, we observed that in experiments where the throughput reached 10 Gbps the *tcpdump* tool did not manage to capture all the observed packets and some of the packets that have been captured were dropped by kernel before being processed by *tcpdump*.

E. Impact on background traffic

In order to study the effects of passive monitoring on background network traffic, we started two application containers and one adjacent monitoring container on each server. The *netperf* tool was used inside the application containers to measure throughput and latency. The traffic between one of the application container pairs was copied to the monitoring container and captured by *tcpdump* while the traffic between the other application container pairs (i.e., background containers) was forwarded normally. We also performed the same measurements without the adjacent monitoring containers where traffic was captured internally inside the containers of one of the application pairs for comparison.

Figure 8 shows the throughput and latency for monitored applications and background applications. It can be seen that for throughput measurements using TCP (with default message sizes of 16384), the monitoring had negligible effect on the background traffic. In RTT measurements, while monitoring increased the latency of the monitored application, the latency perceived by background containers was not affected by using an adjacent monitoring container. Measurements with multiple background container pairs have shown the same results and are therefore not presented here.

F. Scalability

We have evaluated the scalability of our monitoring system by increasing the number of application containers (running *netperf*) which are being monitored. Application containers that belong to the same tenant can share a single monitoring container. However, in a multi-tenant cloud environment, application containers that belong to different tenants require separate monitoring containers to ensure isolation. Therefore, two scenarios were studied: a single monitoring container is used for monitoring multiple application containers (Adjacent-1), and one dedicated monitoring container per application container (Adjacent-N).

Figure 9 shows the total CPU usage on the sender host. It can be seen that the CPU usage is increased by increasing the number of application containers. In scenario Adjacent-1 where a single monitoring container is used, the increase in CPU usage does not change much compared to the Base case, where no monitoring is performed. However, in Adjacent-N scenario where each application container is monitored by a dedicated monitoring container, the increase in CPU usage with increasing the number of containers is slightly higher than running *tcpdump* inside each application container (Internal).

Figure 9 also shows the total memory usage in Gigabytes. It can be seen that by increasing the number of application containers, the memory usage increases. In scenario Adjacent-1 the memory usage is very close to the Base scenario. However, in Adjacent-N scenario the memory usage is higher than the Base but is similar to the Internal scenario.

Additionally, Figure 9 shows that by increasing the number of application containers, the average Throughput is reduced since the application containers have to compete for the 10 Gbps link which is the shared resource. However, it can be seen that the throughput values are not affected much when monitoring is performed even when there is one monitoring container per each application container (Adjacent-N).

VI. Discussion

The evaluation results presented in this paper indicate the feasibility of the ConMon system. It was shown that the most time-consuming step in the automatic setup of monitors in the ConMon system is the creation of monitoring containers and attaching to the virtual switch. This step can be eliminated if monitoring containers are started and attached to the virtual switch in advance. Moreover, if a single monitoring container is used to monitor multiple application containers the CPU and memory usage on the servers is substantially lower compared to using an MF per application container (either internally or in a dedicated adjacent container). These observations suggest

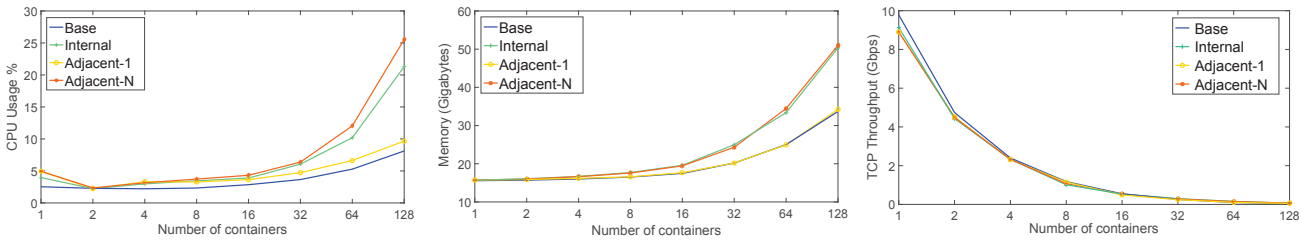

Fig. 9. Scalability results for different number of containers on each host: (left) CPU usage, (middle) Memory usage, and (right) TCP Throughput.

that once a monitoring container is started on a server, it can be reused by updating tapping rules for other application containers.

Our results show that the resource usage overhead of ConMon is manageable. The network overhead, however, is highly depending on the frequency in which the containers are started and stopped. Each MC shares the information about mapping of application and monitoring container(s) with other MCs, for example by updating a distributed database whenever a monitoring container is started or removed. MCs also query the database whenever remote monitor discovery is required. MCs can also communicate with each other to request starting or stopping of remote MFs.

The network overhead required for active monitoring can be high depending on the monitoring intents, e.g., the frequency and type of monitoring. For example, sending one ping packet every second between two physical servers has a negligible overhead while throughput measurements using tools such as *iperf* and *netperf* can introduce higher overhead which can even affect the application and background traffic. ConMon supports reduction of network overhead by reducing the unnecessary overhead by reusing an AM container pair to monitor the network between containers on one server that communicate with containers that reside on the same remote server. However, it is up to the monitoring policies and monitoring functions to implement scheduling strategies to avoid overloading the network links with probe packets.

The accuracy of many types of network measurements, such as latency, depends on accurate timestamping information. The impact of passive monitoring in adjacent monitoring containers on the accuracy of the measured timestamps have been studied in our previous work [20]. In this study we performed additional measurements on a new testbed and observed similar results for different types of traffic which are therefore not presented. Overall, we observed low and stable timestamping errors (in average 3.2 μs and 0.2 μs on the sender and receiver sides, respectively). The errors are mainly caused by the packet processing time of the virtual switch on the sender side and can be used for calibration.

We show that monitoring can affect throughput and latency of the monitored application. However, when applications send traffic with a higher rate than the capacity of the link, the link becomes the bottleneck so the penalty caused by monitoring is not visible. Overall, the results indicate that using an adjacent monitoring container is more suitable than internal data collection within the application containers due to its lower impact on the application performance.

VII. Conclusions

In this paper we presented ConMon, a distributed container-based system for automated monitoring of network performance in a cloud environment.

In ConMon, containers that communicate with each other are automatically discovered and network performance is monitored only between them. The automatic discovery and setup provides uninterrupted monitoring in case of container scaling and migration, without manual intervention. Furthermore, ConMon does not inject monitoring code into application containers and instead executes network monitoring functions inside monitoring containers that are interconnected to the application containers via virtual switches.

By running monitoring functions inside adjacent containers, the monitoring is isolated from the application and does not require instrumenting the image of the application container or running extra processes inside the container. Moreover, the monitoring becomes more flexible since the monitoring functions can be started, stopped, and re-configured either by the application owner or infrastructure providers without affecting the application. Additionally, by passively observing application traffic, the monitoring controllers can quickly adapt the monitoring functions to the changes in the communication between the containers. Therefore, by dynamically adapting to the changes in the container environment and the application communications, ConMon provides continuous monitoring and eliminates the unnecessary cost of monitoring links between the containers that do not communicate with each other.

The evaluation results indicate that the ConMon system is feasible and has negligible impact on background network traffic. However, monitoring does not come without cost and in some cases passive traffic collection can affect the throughput and latency of the applications being monitored. This cost can be reduced by sharing a passive monitoring container to monitor multiple application containers.

References

[1] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization," *ACM SIGOPS Operating Systems Review*, vol. 41, p. 275, 2007.

[2] R. R. W. Felter, A. Ferreira and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *IEEE Symposium on Performance Analysis of Systems and Software*, 2015.
[3] "Nagios." [Online]. Available: https://www.nagios.org
[4] "Amazon CloudWatch." [Online]. Available: https://aws.amazon.com/cloudwatch
[5] "Monasca (Monitoring-as-a-Service (at-Scale))." [Online]. Available: http://monasca.io/
[6] "OpenStack." [Online]. Available: https://www.openstack.org
[7] G. Aceto, A. Botta, W. De Donato, and A. Pescapè, "Cloud monitoring: A survey," *Computer Networks*, vol. 57, no. 9, pp. 2093–2115, 2013.
[8] "Docker." [Online]. Available: https://www.docker.com
[9] "cAdvisor (Container Advisor)." [Online]. Available: https://github.com/google/cadvisor
[10] "Dynatrace." [Online]. Available: https://www.dynatrace.com
[11] "sysdig." [Online]. Available: http://www.sysdig.org
[12] P. Phaal, "sflow version 5, july 2004," 2004. [Online]. Available: http://sflow.org/sflow_version_5.txt
[13] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proceedings of the 2015 ACM SIGCOMM Conference*. ACM, 2015, pp. 139–152. [Online]. Available: http://doi.acm.org/10.1145/2785956.2787496
[14] K. Phemius and M. Bouet, "Monitoring latency with openflow," in *Network and Service Management (CNSM), 2013 9th International Conference on*, Oct 2013, pp. 122–125.
[15] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and H. V. Madhyastha, "Flowsense: Monitoring network utilization with zero measurement cost," in *Proceedings of the 14th Conference on Passive and Active Measurement*. Springer-Verlag, 2013, pp. 31–41. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36516-4_4
[16] C. Fu, W. John, and C. Meirosu, "Eple: an efficient passive lightweight estimator for sdn packet loss measurement," in *Conference on Network Function Virtualization and Software Defined Networks*, ser. IEEE NFV-SDN'16, 2016.
[17] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca, "Planck: Millisecond-scale monitoring and control for commodity networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 407–418, 2015.
[18] W. Wu, G. Wang, A. Akella, and A. Shaikh, "Virtual network diagnosis as a service," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 9.
[19] M. Shahzad and A. X. Liu, "Noise can help: Accurate and efficient per-flow latency measurement without packet probing and time stamping," in *The 2014 ACM International SIGMETRICS Conference*. ACM, 2014, pp. 207–219. [Online]. Available: http://doi.acm.org/10.1145/2591971.2591988
[20] F. Moradi, C. Flinta, A. Johnsson, and C. Meirosu, "On time-stamp accuracy of passive monitoring in a container execution environment," in *2016 IFIP Networking Conference*, 2016, pp. 117–125. [Online]. Available: http://dx.doi.org/10.1109/IFIPNetworking.2016.7497236
[21] K. Hedayat, R. Krzanowski, A. Morton, K. Yum, and J. Babiarz, "A Two-Way Active Measurement Protocol (TWAMP)," RFC 5357 (Proposed Standard), Internet Engineering Task Force, Oct. 2008, updated by RFCs 5618, 5938, 6038, 7717, 7750. [Online]. Available: http://www.ietf.org/rfc/rfc5357.txt
[22] "iPerf." [Online]. Available: https://iperf.fr
[23] "Netperf." [Online]. Available: www.netperf.org
[24] P. Naik, D. K. Shaw, and M. Vutukuru, "Nfvperf: Online performance monitoring and bottleneck detection for nfv," in *Proceedings of the IEEE Conference on Network Function Virtualization and Software Defined Networks*, 2016.
[25] R. R. Kompella, K. Levchenko, A. C. Snoeren, and G. Varghese, "Every microsecond counts: Tracking fine-grain latencies with a lossy difference aggregator," in *Proceedings of the ACM SIGCOMM 2009 Conference*. ACM, 2009, pp. 255–266. [Online]. Available: http://doi.acm.org/10.1145/1592568.1592599
[26] "pipework." [Online]. Available: https://github.com/jpetazzo/pipework
[27] "Consul." [Online]. Available: https://www.consul.io
[28] "System Activity Report (SAR)." [Online]. Available: http://linux.die.net/man/1/sar