

Towards a Client-Centric QoS Auto-Scaling System

Thomas Lin and Alberto Leon-Garcia
Department of Electrical and Computer Engineering
University of Toronto, Toronto, ON, Canada
Email: {t.lin, alberto.leongarcia}@utoronto.ca

Abstract—Many modern day cloud services are composites of multiple smaller services working correctly together. This design has become increasingly prevalent due to the rise of the microservices application architecture, as well as service chaining in Network Function Virtualization (NFV). Future composite applications and services will be deployed on multi-tier clouds where their constituent microservices may be geographically spread over different regions. To optimize the delivery of such composites, the constituent microservices must be placed in locations where their clients, which may be other microservices, are able to meet certain QoS constraints. We propose an architecture and present a prototype system for incorporating network metrics into the auto-scaling and scheduling decisions of cloud management systems. Given a service with QoS constraints, our system monitors the network metrics (e.g. latency and bandwidth) of their clients. If a particular client is unable to receive the required latency or bandwidth of the service, our system auto-scales the service and strategically places the new instance(s) in a location capable of meeting the service quality, and re-directs traffic to the new instance.

I. INTRODUCTION

Within the last decade, the growing trend of the microservices architecture, a method for fine-grained decomposition of traditionally monolithic software into individual intercommunicating “microservices”, has increased the complexity of modern cloud applications and services. In these complex systems, performing a single task may involve a series of sequential calls between different microservices. As an example, the call-flow to boot a single virtual machine (VM) within the OpenStack¹ architecture requires a series of at least 28 different sequential operations [1] performed by various services. In a similar vein, in the Network Function Virtualization (NFV) architecture, building blocks of virtualized network functions (VNFs) can be orchestrated into “service chains” [2] to create new composite network services. In both these architectures, the distinction between “clients” and “services” has become blurred, in that a service instance can itself be the client of another service. Thus, the definition of a service is dependent on context; it can be either an end-service (e.g. a database), or a middle-box (e.g. a network function) along the path to an end-service.

The performance of composite applications and services designed using these architectures often depends on whether the individual services (microservices or VNFs) are able to effectively serve each other’s requests. Each service may have

its own definition as to what network quality of service (QoS) metrics (e.g. latency and bandwidth) their clients require to seamlessly receive the service. Thus, the careful placement of these services can have a big impact on the performance on the composite application or service, especially if it plays a critical role within the realm of smart cities, power grids, traffic management, etc.

The common design pattern for realizing service guarantees in modern clouds is the use of auto-scaling to guarantee capacity. This typically uses a monitoring system that has the ability to trigger a scaling operation on the service to create new instances. Cloud network monitoring solutions traditionally focus on measuring the bandwidth utilization of the service instances, and auto-scales to ensure individual instances are not overloaded. This type of QoS guarantee is focused on individual service instances under the premise that if the instances have enough network capacity, then the clients will receive an optimal service. We argue that QoS metrics such as bandwidth and/or latency can and should be defined from the point of view of the client. In other words, defining a network QoS should be akin to asking the question: what are the minimal network QoS requirements (e.g. bandwidth and latency) for a client to receive this service seamlessly? Cloud monitoring and auto-scaling solutions should then strive to meet these requirements for individual clients.

When services auto-scale, they rely on a scheduler to decide which node within a cluster to place the new service instance. The default schedulers in popular open-source systems such as OpenStack or Kubernetes² often make simple decisions based on the available CPU and memory of each node. Consequently, critical metrics, such as the bandwidth and latencies of clients that consume a particular service, are often overlooked. Incorporating network metrics into the scheduling decision enables us to strategically place service instances to improve the QoS of their clients, and further, enables improving the performance of composite application or service by dynamically adjusting their placements.

The ability to adaptively place services closer to clients is particularly important as we are seeing hybrid-cloud architectures where multi-tier applications sit atop a mix of both on-prem clouds and public clouds. In the emerging IoT era, there may also be physical hosts (e.g. IoT gateways or mobile customer premise edges) that connect, disconnect, and change locations, resulting in dynamic infrastructure topologies that

¹OpenStack: <https://www.openstack.org/>

²Kubernetes: <https://kubernetes.io/>

require re-placement of service instances. Other direct and indirect benefits include:

- Reduced service latency experienced by clients;
- Reduced bandwidth wastage: Naïve scheduling may place a new service instance on a node with plenty of free CPU and RAM, but which may be located far away from the most demanding clients, such that the clients' traffic must traverse many links to reach the new replica; and
- Improved service availability when temporary link outages cuts off part of the core network.

In this paper, we propose an architecture and present a prototype capable of monitoring the network service quality on a per-client basis. The monitoring information is used to detect if a client's pre-determined QoS has been violated, and auto-scales if necessary. The monitoring information is further used to inform the cloud scheduler's placement decisions. Network control actions can be exercised to re-direct traffic from client(s) to the new service instances to improve their QoS. The rest of this paper is organized as follows: In Section II we review background and related works. Section III introduces our design and high-level architecture, while Section IV describes our prototype implementation. We present our evaluations in Section V and finally, Section VI concludes this paper and discusses future work.

II. BACKGROUND AND RELATED WORKS

In this section, we first present an overview of auto-scaling in distributed cloud computing environments and some of their challenges. We then present other works that have attempted to improve network metric collection in SDN-enabled networks and to integrate network information for improving scheduling and placement.

A. Auto-Scaling in Clouds

Auto-scaling in cloud environments remains an ongoing and active area of research. A simple deployable solution consists of a monitoring component to watch the utilization of a given set of services and a resource controller to boot new service instances if the utilization goes up, or to delete service instances if the utilization goes down. Ceilometer³, the monitoring component of OpenStack, is an early open-sourced solution that enabled auto-scaling with the use of threshold-based alarms that can be tied to a web-hook (a URL that will be called when the alarm is raised or lowered).

More recently, containers and the rise of the microservice architecture has prompted new and more complex tools to be developed. In [3], the authors developed a tool around Docker Swarm and OpenStack that is able to jointly orchestrate and scale both "macroservices" (i.e. VMs) and "microservices" (i.e. containers), based on monitoring information provided by the ELK stack⁴. The Kubernetes container orchestration system comes with a Horizontal Pod Autoscaler [4] for horizontally scaling a service based on observed CPU and memory

utilization. In addition, the system also supports a Vertical Pod Autoscaler to adjust the CPU and memory allocations per pod as their monitored utilization increases or decreases. Pods scaled by the horizontal autoscaler are placed according to which node has the available CPU and memory to run the new instance; and similarly, the vertical autoscaler will evict pods and schedule a new one if the current physical node cannot meet the updated CPU or memory requirements. In their current default form, the autoscalers are threshold-based with scaling and placement decisions based on simple compute metrics (i.e. CPU and RAM utilization) [5]. It is worth noting that the Kubernetes community is currently working on a custom metrics API to enable non-conventional metrics to be used.

B. SDN-Assisted Network Monitoring

With the advent of distributed microservices, factoring in the network state in the optimization of the scheduling of instances becomes an area of research that deserves more attention [6]. However, even in SDN-enabled environments, it remains a challenge to collect full information of the network such as bandwidth and latency. To this end, the authors of [7] proposed to augment the SDN controller's view of the network by providing it updates from an external network management system (NMS).

In [8], the authors propose an architecture to enable in-band network telemetry (INT) as a means towards realizing a "self-driving network" based on knowledge-defined networking [9] (KDN). To enable the self-driving network, the SDN controller configures the data plane switches to support INT packets to gather data for the knowledge plane, where actions requiring knowledge (e.g. resource planning, optimization for performance, etc.) are handled.

C. Network-Aware Scheduling

One of the earliest implementations of a network-aware scheduling system was presented in [10], where the authors augmented OpenStack to take network constraints into consideration when placing VM instances. To avoid patching existing code in Nova and Neutron (the compute and network components of OpenStack, respectively), the authors added bandwidth information to the "extra specs" section of instance flavours. An external process then queries the VMs on a given physical compute node, sums the bandwidths of each VM's flavour to determine the current allocation, and determines the compute node's currently available bandwidth by subtracting the allocation from the node's max bandwidth. However, this method does not measure the actual bandwidth being used, only what is set in a flavour; and in addition, this method is unable to identify cases where network bottlenecks appear in inter-switch links within the topology (i.e. not the last hop).

In [11], the authors extended OpenStack to enable network-aware placement of virtual network functions (VNFs) in a multi-tier cloud deployment. Using VMTP [12], an open-source tool developed by Cisco for OpenStack, they collect information regarding the available network (bandwidth and

³OpenStack Ceilometer: <https://docs.openstack.org/ceilometer/latest/>

⁴ELK Stack: <https://www.elastic.co/what-is/elk-stack>

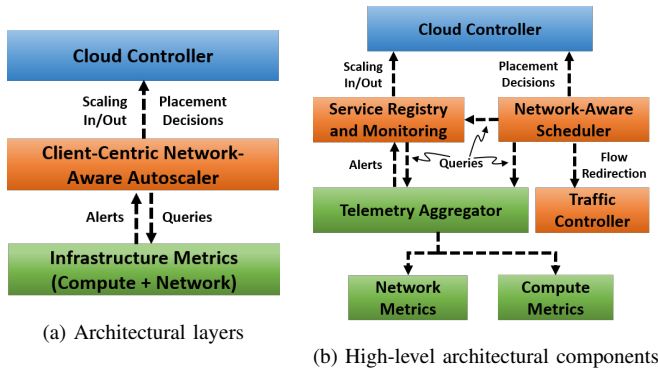


Fig. 1: Generalized architectural layers and high-level breakdown of required components.

latency) resources. However, VMTP adds extra overhead as it needs to SSH into the underlying physical host to perform the bandwidth and latency measurements. Thus, this tool assumes the physical host’s management and data networks are the same (i.e. in-band). Their orchestration algorithm, DARK, then uses the collected network information to determine placement and ensures the location meets both the compute requirements of the VNF, as well as the latency and bandwidth requirements of the overall service chain.

Network-aware scheduling isn’t limited to the placement of compute resources, but can also be used to optimize job-based schedulers as well. In [13], the authors present a bandwidth-aware task scheduler for Hadoop clusters. They propose a formulation where bandwidth information, collected using an SDN controller, is fed into BASS, their proposed task scheduling algorithm. Their evaluation showed that bandwidth-aware task scheduling decreases job completion times compared to Hadoop’s default scheduler.

III. HIGH-LEVEL ARCHITECTURE

In this section, we present our design and architecture for enabling client-specific network monitoring and service optimization. Figure 1a shows a generalized high-level architecture of the layers involved in realizing our goal, where most of our work is focused on the middle layer. Figure 1b shows a component breakdown of our high-level architecture, where the colours correspond to the similarly coloured layer in Fig. 1a. We begin by discussing the requirements of the target environment which our framework is designed for, and later, we will discuss the various components from Fig. 1b.

A. Environment Requirements

We assume the cloud environment that our architecture is designed for supports virtualized compute resources, either VMs, containers, or both. In fact, we do not distinguish the actual type of compute resource, so long as they are network-connected and have some type of globally unique network address. This is a critical requirement for dynamic traffic re-direction.

To perform the traffic re-direction, we require that the cloud environment support traffic control capabilities for dynamically altering network flows. Ideally, the cloud’s control and

management plane exposes a set of network-related APIs for other authorized components to leverage the capabilities to set up new (or change existing) flows. In addition, such APIs should allow the collection of statistics and other information from the network. In environments where it is either difficult or impossible to natively support such capabilities (e.g. multi-operator hybrid deployments, legacy systems, etc.), SDN-enabled overlays can be used to achieve the same capabilities at the cost of throughput when passing high-volume traffic to and from the underlying network [14].

While our goal is to collect the network information for the scheduler, the placement decision should not be performed based on this information alone. The scheduler should already have access to a breadth of compute-related metrics such as CPU, RAM, disk I/O, etc., as well as infrastructure state information from a cloud-aware monitoring system. To elaborate, we define cloud-awareness to mean the system is cognizant that resources in the infrastructure may be virtualized, with active monitoring of both physical and virtual resources, as well as tracking the relationships between virtual-to-physical.

B. Service Registry and Monitoring

The service registry enables users to register and to track particular services. Since a service is a logical unit that may be composed of one or more instances, we must update this logical unit as instances are created or removed. Furthermore, the service registration should allow the users to define: 1) the traffic type of the service, either stateless or stateful; and 2) the service quality metrics its clients require.

Specifying the traffic type (i.e. stateless or stateful) is critical because services that rely on state-based protocols should not have their established connections be dynamically re-directed to new instances, as this may effectively break the connection if the service is not designed for stateful scaling (though this is possible using the works such as [15] or [16]). Newly initiated connections, however, are considered safe to re-direct. If a service uses a stateless protocol (e.g. DNS queries), then we assume it is safe to dynamically re-direct existing clients.

The service quality metrics define the QoS required by the clients in order to properly consume the service. These may include network requirements such as maximum latency tolerance, guaranteed minimum bandwidth, jitter, and etc. As the infrastructure and network management systems improve to support control and fine-tuning of other network performance metrics (e.g. packet loss tolerance), these metrics can be included in the quality definition.

Using the service registry information, the system is able to track and monitor the individual instance replicas of a particular service. From the cloud controller, the system is also able to obtain detailed information on each replica; in particular, which physical node they are hosted on, as well as their globally unique network address. This information is used to dynamically enable network monitoring of flows going to and from the various instances of a particular service, which allows per-client network metrics to be collected and stored by the telemetry aggregator. The data within the telemetry

aggregator can further be used for long term analysis of the service's usage trends.

Having per-client network metrics augments the traditional compute-related monitoring information that most auto-scaling systems rely on. This enables strategies such as identifying the most offending clients when a service instance is under heavy load, and selectively re-directing them to other instances. Additionally, this information enables the infrastructure management system to determine if a client is unable to receive the service's defined network quality, and to remedy it by placing a new instance at a location that is better able to meet the client's required service quality.

C. Network-Aware Scheduler

When a new instance of a service is to be launched, the network-aware scheduler is involved in deciding where to place it. To make its placement decision, the scheduler consults: 1) the service registry to obtain the required quality metrics of the service; and 2) the telemetry aggregator to identify a physical node that can meet the quality metrics of the service. The scheduler takes into account both the compute and network requirements of the service to make its decision. Ideally, the scheduler should be pluggable, enabling different network-aware scheduling algorithms to be used. The actual scheduling algorithms and heuristics, however, are out-of-scope of the current paper. Once the scheduler informs the cloud controller where to place the new instance, it then notifies the traffic controller that a new instance of the service is going to be placed there.

One major difference when performing client-centric network-aware scheduling is that the placement decision is made relative to some reference point in the network. For example, suppose a service is latency and bandwidth constrained. To service a client outside the network domain, the latency and available bandwidth is calculated for the path from the ingress gateway to the new instance. However, if the instance is part of a service function chain, then the latency and available bandwidth is calculated for the path from some previous function within the chain.

D. Traffic Controller

Once a new service instance has been created and the network-aware scheduler makes its placement decision, it alerts the traffic controller to modify the traffic flows of the network, re-directing the traffic of new client connections to the new instance. If the service was defined as stateless, it can also re-direct existing clients, who were unable to receive the required QoS, to the new instance.

In the stateless case, the decision regarding which client's traffic to re-direct depends on what originally triggered the auto-scaling operation. If the trigger condition was due to a service instance being congested, the client using the most resources on that instance can be re-directed to the new instance. On the other hand, if the trigger condition was due to a client unable to receive its required service quality, then that is the client to be re-directed. The re-direction may not

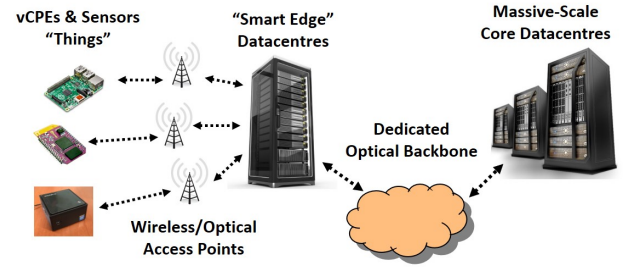


Fig. 2: SAVI multi-tier deployment

be limited to just the flow of a single client, it can re-direct a set of existing flows to the new instance. This could be done, for example, if the traffic controller attempts to re-balance the load on the overall service by calculating which other clients of existing instances would be better serviced by the new instance.

To achieve the re-direction, the system requires a software-based networking approach that can be re-configured on the fly. We define *software-based* networking as being any type of network control implemented using software, which could include, through not necessarily, traditional software-defined networking. For example, software proxies such as Envoy⁵, software routers such as OpenStack Neutron, or more complex systems such as services meshes like Istio⁶, are all software-based networking solutions.

IV. IMPLEMENTATION

We present an initial implementation of our architecture with a focus on the network monitoring components, and we introduce a basic traffic re-direction mechanism. We begin by describing the cloud platform our work is built upon, then we give an overview of the components used to realize our architecture. Afterwards, we discuss our methodology for collecting client-based latency and bandwidth.

A. SAVI Testbed

We implemented and tested our system on the Smart Applications on Virtualized Infrastructure (SAVI) testbed [17]. The SAVI testbed has been in operation since 2012 and is a multi-tier and multi-tenant cloud. The testbed spans multiple regions that are geographically spread throughout Canada. Its multi-tier architecture, shown in Fig. 2, enables the strategic deployment of application sub-components based on resource and service constraints. SAVI's networking is fully SDN-enabled using OpenFlow [18] switches, both physical and software, and its control and management plane is based on our concept of Software-Defined Infrastructure (SDI) [19]. Our SDI-based control plane has a centralized graph-based view of the infrastructure topology, for both the compute and network resources, which enables global planning and optimization.

B. Component Overview

Our current implementation of the architecture is shown in Fig. 3, where the colours correspond to the similarly

⁵Envoy Proxy: <https://www.envoyproxy.io/>

⁶Istio: <https://istio.io/>

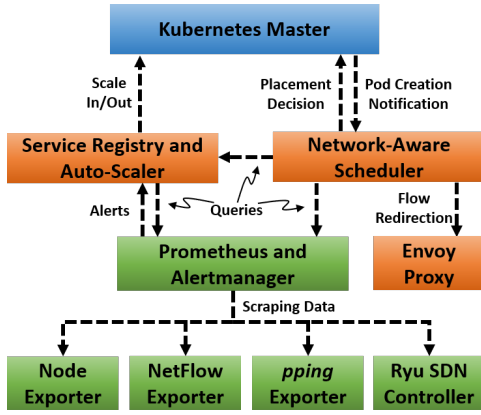


Fig. 3: Components of implemented architecture

coloured layer from Fig. 1a. For the cloud controller, we use Kubernetes, where we work with pods as our compute resources. Our SDN-enabled network uses OpenFlow, and so we use Ryu⁷ (a Python-based OpenFlow controller) to control our network.

To collect and store our telemetry data, we deploy Prometheus⁸ as a database, and Alertmanager, its sister component for sending alerts. For collecting compute metrics, we use the Prometheus Node Exporter to expose OS and hardware metrics from the underlying compute nodes of our cluster. To get container information for the cluster, we scrape cAdvisor⁹, a container performance and monitoring agent that comes integrated within Kubernetes agent binaries. The network-related metrics are collected from: 1) a custom-made NetFlow collector; 2) a client latency measurement component; and 3) the Ryu controller that runs custom-made applications to measure inter-switch link latency.

The service registry, NetFlow collector, client latency component, network-aware scheduler, and the applications running on Ryu, were all newly developed for this work. The current implementation of the service registry system supports maximum latency tolerance and guaranteed minimum bandwidth as the service quality metrics. The current traffic control mechanism leverages Envoy for stateful traffic re-direction.

C. Inter-Switch Link Latency

Monitoring inter-switch link latency allows us to determine the latency of any end-to-end path that begins and ends within our network topology. More importantly, it allows the system to quickly root-cause the offending link(s) when it detects that a client's end-to-end latency has unexpectedly increased. Network control actions can be taken to route flows around the affected link(s). Additionally, this enables the network-aware scheduler to place new instances in nodes downstream from the affected links.

There are many factors that impact the link latencies in a network: propagation delay, queueing delay, and switch

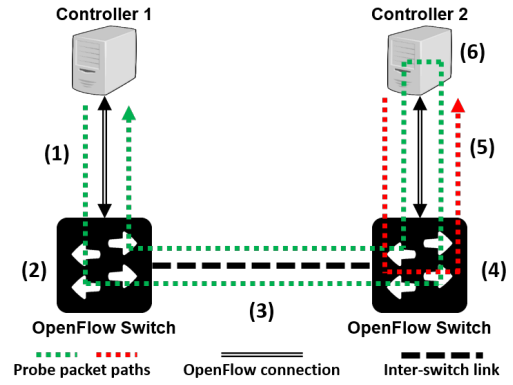


Fig. 4: SDN-based link latency estimation.

Our technique estimates the link latency (3) using probes that enable us to account for the OpenFlow connection delay (1 & 5), OpenFlow table look-up delay (2 & 4), controller processing delay (6).

processing delay. Complicating the problem is the fact that the performance of software-based switches such as Linux Bridge or Open vSwitch¹⁰ vary depending on the CPU model and clock, as well as the overall system load, of the host they run on. This becomes more complex in cloud environments when the system load of servers vary over time depending on the guests' application workloads.

To account for the effects of all these factors, we have developed an SDN controller-based latency estimation technique that works with independently distributed controllers. Due to paper length constraints, we provide only a brief overview. Our technique, which is currently implemented in an OpenFlow application deployed on Ryu, periodically sends two different types of probes generated by individual controllers (see Fig. 4). The first probe type is sent to the controllers of adjacent switches (which may not be the same controller), who then return the probes to the originating controller. The second probe type is sent from controllers to their connected switches, where a flow table entry sends it back to the controller. These probes allow us to simultaneously capture the delay induced by various factors:

- The controller packet processing time;
- The OpenFlow connection latency;
- The OpenFlow switch table lookup time; and
- The switch queuing and propagation delay (i.e. link latency);

The controllers continuously probe these factors over time to watch for changes that affect the networking latency. As these probed metrics may be noisy in cloud environments with other active workloads, we smooth out the latency estimates using a running median filter, where the window is adjustable.

A secondary Ryu application has also been developed to export the latency metrics to Prometheus. This exporter application applies more aggressive smoothing by combining an exponential moving average (EMA) with the moving median. We use the EMA (currently set with a smoothing factor of 0.125) to prevent large step sizes, and gradually adjust to new

⁷Ryu: <https://osrg.github.io/ryu/>

⁸Prometheus: <https://prometheus.io/>

⁹cAdvisor: <https://github.com/google/cadvisor>

¹⁰Open vSwitch: <https://www.openvswitch.org/>

trends over time. However, given a large enough temporary surge in latency, even with the EMA it is still possible to get a large temporary step. We use the moving median, currently set to a window size of 30, to compensate for these situations. Thus, any persistent changes in the trend must persist for at least 15 continuous samples before our system begins to register it. All of these parameters are tunable by the system administrator, depending on how conservative they want the alerting to be.

D. Client Service Latency

Low-latency applications are a key motivator for the proliferation of edge computing. Monitoring the end-to-end latency experienced by clients allows the service registry to auto-scale a service if it detects that a client is not receiving the required service quality, and allows the network-aware scheduler to find a candidate node that can meet the desired service quality. There are two scenarios to consider, each with a different approach to calculating the service latency:

The client is within our network domain: If the client is within our network, then the service latency can be calculated by adding the link and switch processing latencies along the path from the client to the service instances. This scenario can arise either in the case of a regular client, or in the case of a node within an NFV service chain.

The client exists outside our network domain: If the client exists outside our domain of control and uses a TCP-based protocol, we use *pping* [20] to passively snoop the traffic right before it enters the service instance, and collect packet metadata to calculate the latency to the client. We have created a Prometheus exporter around *pping* to enable the analysis and collection of latency data. Given the large number of endpoints in our system, we configure *pping* to only track the endpoints corresponding to the services we are tracking. On the other hand, if the protocol is not based on TCP, then the best latency estimate we can currently provide is the latency to the ingress gateway where the traffic enters our network.

E. Client Service Bandwidth

All network switches within the SAVI testbed support exporting of NetFlow data, which is traditionally streamed to a NetFlow collector for long-term storage, trending, and analysis. We have implemented a custom NetFlow collector that also serves as a Prometheus exporter. Our custom NetFlow collector expires inactive flows after a period of time (a tunable parameter, currently set to 10 minutes in our deployment) to prevent polluting the database with stale flows. Analyzing the NetFlow information provides us the bandwidth utilization for particular clients, as well as for individual links within the network.

F. Traffic Redirection

When the scheduler places new services instances, it is also responsible for informing the proxy component(s) to redirect traffic from clients (either new or existing connections) towards the new instance. Our current implementation utilizes

Envoy because it supports fine-grained proxy rules that can match specific source IPs and direct them to specific service instances. More importantly, it supports live reconfiguration (via APIs or a discovery service) without disrupting existing TCP connections. This feature allows clients with ongoing transactions with services to gracefully finish, and subsequent connections from the same clients will be re-directed to the new instance.

G. Alert Call Flow

Since Prometheus aggregates all the metrics, the service registry is able to dynamically configure it to watch for service quality violations (e.g. latency of a connection goes over 10 ms) and alert the registry and auto-scaler component. Referring back to Fig. 3, we summarize the call flow between components when an alert is triggered and a scale-out operation is desired:

- 1) The service registry and auto-scaler component decides what to do based on the received alert (e.g. scale out or wait), querying Prometheus for extra data if needed. If scaling, it calls the Kubernetes master to create a new pod (the basic unit of compute in Kubernetes);
- 2) The Kubernetes master notifies the network-aware scheduler if a new pod needs scheduling;
- 3) The scheduler queries Prometheus for the required data to make its placement decision, as well as the service registry to see what constraints the pod must meet;
- 4) The scheduler's placement decision is communicated back to the Kubernetes master; and finally
- 5) The scheduler queries Prometheus for active alerts, and updates the proxy service to redirect traffic from clients (that triggered the alert) to the new pod.

If the desired operation of an alert is to scale in, the call flow is a bit different. Due to limitations with the current iteration of Kubernetes, the scheduler is not consulted for pod deletions, and Kubernetes simply chooses the pod with least uptime. To work around this limitation, our auto-scaler component queries Prometheus to determine the pod that least meets the desired service quality metrics, and explicitly deletes that pod. Kubernetes, to keep the number of pods the same, reacts by creating a replacement pod. The auto-scaler then issues a scale-in operation, and the new replacement pod will be deleted due to its small uptime.

V. EVALUATION

In this section, we present a set of experimental evaluations to functionally validate our architecture.

A. Environment

We first begin with a description of our test setup on the SAVI testbed. Within an isolated tenant, we deploy a multi-tier Kubernetes cluster using virtual machines spread out over two regions of the testbed (Waterloo and Toronto). The two regions are inter-connected by a dedicated optical backbone network. The VM specifications are shown below in Table I, and our cluster's topology is shown in Fig. 5.

TABLE I: Kubernetes Cluster Specifications

Server	RAM (GB)	Intel Xeon CPU	# of Cores
K8s Master	8	E5-2650 v2	8
K8s Nodes	4	E5-2650, E3-1230 v3	2
SDN Controller	4	E5-2650 v4	2
Monitoring & Alerting	16	E5-2650 v4	16
Gateway	2	E5-2650	1

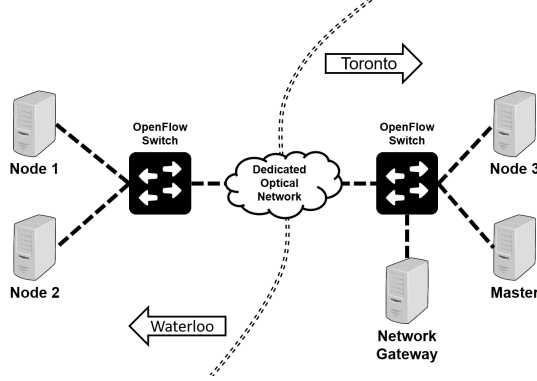


Fig. 5: Cluster Topology

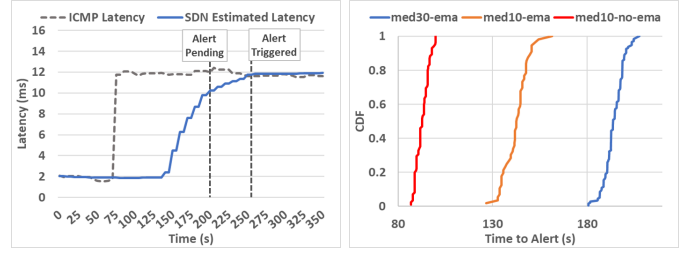
B. Latency Monitoring Reaction Time

To determine how quickly our system is able to react to latency changes in the underlying network, various factors that affect the metrics collection system must be considered. Our implementation, which combines Prometheus, Alertmanager, and the SDN-based latency estimator, involves many admin-defined settings that can influence the detection time; from the bottom-up, these are:

- 1) The sampling interval of our SDN latency estimator;
- 2) Smoothing methods used in the SDN latency estimator (see subsection IV-C);
- 3) Prometheus' scrape interval for the target (i.e. the SDN controller);
- 4) Prometheus' alert evaluation interval;
- 5) Prometheus' alert "for" clause (i.e. minimum time the alert condition must persist before informing Alertmanager); and
- 6) Alertmanager's "group_wait" clause (i.e. minimum time to wait for a similar notifications from Prometheus before sending an alert; used to prevent duplicate alerts).

Depending on how the intervals and wait times align, as well as when events occur, the system reaction time may vary.

We design an experiment in which we artificially increase the latency to a service in Node 3 and measure the delay until the system triggers an alert. In our setup, the SDN latency estimator's sampling interval is 5s, the running median window is 30 samples, Prometheus' scrape interval is 10s, its alert evaluation interval is 5s, its "for" clause is 1m, and Alertmanager's "group_wait" clause is 0s (i.e. immediate alert). While Prometheus can support scrape and evaluation intervals of 1s, we choose this set of interval parameters to provide an optimal balance between timely alerts and reducing the resource overhead consumed by the monitoring system. As



(a) Latency from the Gateway to Node 3
Actual vs SDN-based estimate

(b) CDF of the system's reaction time to changes in latency.
Three different smoothing techniques used by SDN-based estimator.

Fig. 6: Latency-based alert (all parameters are tuneable).
Alert condition: latency > 10 ms

the goal of this system is to capture trends over time, missing microbursts of latency between scrapes is tolerable.

Figure 6a shows the increase in latency at ~70s, but the running median filter delays detection until later at ~140s. Our EMA filter and the 1m "for" clause further delays the alert until ~255s. The blue line (med30-ema) in Fig. 6b shows a cumulative distribution function (CDF) of the reaction time under these configurations, with an average and median of roughly 194s. The step-wise nature of the curve is reflective of the fact that all the components of our system use some type of interval-based sampling.

Our current configuration settings are specifically designed to be conservative. If the network latency changes only temporarily, we wish to avoid sending out false-positive alerts to trigger unnecessary scaling operations and thus wasting compute resources. System administrators willing to tolerate such scaling operations can tune the parameters to react faster. Figure 6b also shows the CDFs when we shrink the median window to 10 samples (med10-ema), and if we further remove the EMA filter (med10-no-ema). In the med10-no-ema case, the delay is mostly due to the Prometheus "for" clause.

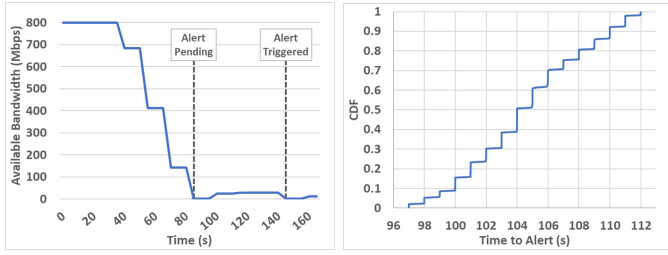
C. Bandwidth Monitoring Reaction Time

We now consider how quickly our system reacts to bandwidth changes in the network. We design an experiment involving a node where a service instance exists, and define its guaranteed minimum bandwidth to be 50 Mbps. We then reduce the service instance's available bandwidth by congesting the network with traffic, and measure the delay until an alert is triggered.

Our switches are configured to export NetFlow data at most every 5s. Prometheus is configured to scrape the NetFlow exporter at 5s intervals, with an alert evaluation interval of 5s, a "for" clause of 1m, and Alertmanager's "group_wait" clause is set to 0s (i.e. immediate alert). Figure 7a shows the reaction time, where the congestion starts at ~40s and the alert is triggered at ~145s. Figure 7b shows a CDF of the reaction time, with a median and average of roughly 104s.

D. System Overhead

For the system components we implemented, we monitor their resource utilization to determine the resource overhead.



(a) Available bandwidth of service instance (b) CDF of the system's reaction time to a reduction of available bandwidth.

Fig. 7: Bandwidth-based alert (all parameters are tuneable). Alert condition: Service's available bandwidth < 50 Mbps

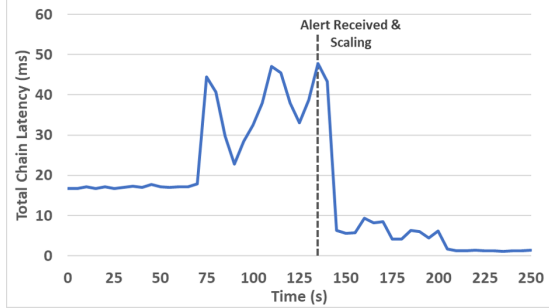


Fig. 8: Total latency from gateway through middlebox to web server.

Specifically, we measure the average CPU (per-core), memory, and network utilization for over a day. The results are shown in Table II.

TABLE II: Overhead of Monitoring Components

Component	RAM (MB)	CPU (%)	Bandwidth (kbps)
Ryu Latency Exporter	133.84	0.92	5.81
NetFlow Exporter	46.13	3.27	17.62
pping Exporter	23.02	13.91	16.42

The *pping* exporter result is for a single instance in a node with active service instances. The higher CPU overhead seen for the *pping* exporter is due to its current implementation using Python. Re-implementing it in a language such as Go or C/C++ would drastically reduce its utilization, which is important as we deploy *pping* on a per-node basis. Conversely, the latency exporter on Ryu has a low CPU overhead due to the fact that we implemented our estimation technique in C++, and only used Python as a thin wrapper to expose the data. Finally, the NetFlow exporter (implemented in Go) result is for a two-switch network, and is shown to have low overhead; however, its bandwidth and CPU utilization is expected to grow relative to the number of switches in the network. In deployment, a single NetFlow exporter can handle the data from multiple switches.

E. Service Chain Example

We demonstrate the overall functionality of the system by presenting a simple chain involving a middlebox service and a web server end-service. Referring back to Fig. 5, we have placed the middlebox in Node 2 (Waterloo) and the web server

in Node 3 (Toronto). The middlebox service initially meets its latency QoS, defined as latency (from the gateway) under 10ms. Figure 8 shows the total end-to-end latency (gateway to middlebox to webserver) during our experiment. At ~75s, we congest the link between the two switches, resulting in an increased latency from the gateway to the middlebox. This increase causes an alert at ~135s as the middlebox service's QoS has been violated, and forces the system to scale and place a new instance at Node 3, followed by a redirection of traffic to the new instance. Since both the new middlebox instance and the web server are now within the same region, the latency is drastically reduced. From the time of alert, it took less than 10s for the latency violation to be mitigated.

With the existing system implementation, we can support diverse applications and services. Some other examples include: VPN-as-a-Service, which are generally both latency and bandwidth constrained; intrusion prevention systems, which are often bandwidth constrained; distributed DNS, which is latency constrained; or some combination of all three.

VI. CONCLUSION AND FUTURE WORK

Traditional cloud auto-scaling and scheduling solutions often do not consider the required network QoS of clients when scaling and placing service instances. This is particularly important in the modern era where cloud applications are increasingly designed using the microservices architecture, and cloud operators are adopting NFV service chains. The performance of such composite applications and services, that involve many inter-communicating services, often depend on the placement of each service relative to each other.

In this paper, we proposed an architecture and implemented a prototype capable of: 1) monitoring the QoS of clients connected to services; 2) auto-scaling when a client's QoS has been violated; 3) performing network-aware scheduling to place new instances at locations that can better service the clients (e.g. lower latency, more bandwidth); and 4) statefully re-directing client traffic to those new instances. We deployed our prototype on the SAVI testbed, a multi-tier cloud, and validated its ability to detect client QoS violations, place new service instances relative to the client, and re-direct traffic.

This work presents an important step towards enabling novel orchestration algorithms to perform dynamic placement optimizations of multi-tier composite applications and services. We leave the exploration and integration of such algorithms as future work. As the SAVI testbed is equipped with physical OpenFlow switches, another aspect we intend to investigate as future work is the implementation of stateful traffic redirection directly in OpenFlow, as that would give us access to full line-rate.

REFERENCES

- [1] R. Gupta, "Request Flow for Provisioning Instance in Openstack." [Online] Available: <https://ilearnstack.wordpress.com/2013/04/26/request-flow-for-provisioning-instance-in-openstack/> [Accessed: August 2019].
- [2] ETSI, "Network Functions Virtualisation (NFV); Ecosystem; Report on SDN Usage in NFV Architectural Framework," in *ETSI GS NFV-EVE 005 V1.1.1*, 2015.

- [3] H. Khazaei, R. Ravichandiran, B. Park, H. Bannazadeh, A. Tizghadam, and A. Leon-Garcia, "Elastic: Autoscaling and Monitoring As a Service," in *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*, CASCON '17, (Riverton, NJ, USA), pp. 234–240, IBM Corp., November 2017.
- [4] "Horizontal Pod Autoscaler." [Online] Available: <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/> [Accessed: April 2019].
- [5] "Resource metrics pipeline." [Online] Available: <https://kubernetes.io/docs/tasks/debug-application-cluster/resource-metrics-pipeline/> [Accessed: April 2019].
- [6] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari, "Open Issues in Scheduling Microservices in the Cloud," *IEEE Cloud Computing*, vol. 3, pp. 81–88, Sep. 2016.
- [7] S. Lange, L. Reinhart, T. Zinner, D. Hock, N. Gray, and P. Tran-Gia, "Integrating Network Management Information into the SDN Control Plane," in *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–9, April 2018.
- [8] J. Hyun, N. Van Tu, and J. W. Hong, "Towards Knowledge-Defined Networking using In-band Network Telemetry," in *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–7, April 2018.
- [9] A. Mestres, A. Rodriguez-Natal, J. Carner, P. Barlet-Ros, E. Alarcón, M. Solé, V. Muntés-Mulero, D. Meyer, S. Barkai, M. J. Hibbett, G. Estrada, K. Ma'ruf, F. Coras, V. Ermagan, H. Latapie, C. Cassar, J. Evans, F. Maino, J. Walrand, and A. Cabellos, "Knowledge-Defined Networking," *SIGCOMM Comput. Commun. Rev.*, vol. 47, pp. 2–10, Sept. 2017.
- [10] M. Scharf, M. Stein, T. Voith, and V. Hilt, "Network-Aware Instance Scheduling in OpenStack," in *2015 24th International Conference on Computer Communication and Networks (ICCCN)*, pp. 1–6, Aug 2015.
- [11] D. Haja, M. Szabo, M. Szalay, A. Nagy, A. Kern, L. Toka, and B. Sonkoly, "How to orchestrate a distributed OpenStack," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 293–298, April 2018.
- [12] "Addressing data plane performance measurement on OpenStack clouds using VMTP." [Online] Available: <https://blogs.cisco.com/cloud/addressing-data-plane-performance-measurement-on-openstack-clouds-using-vmtp> [Accessed: April 2019].
- [13] P. Qin, B. Dai, B. Huang, and G. Xu, "Bandwidth-Aware Scheduling With SDN in Hadoop: A New Trend for Big Data," *IEEE Systems Journal*, vol. 11, pp. 2337–2344, Dec 2017.
- [14] Sandhya, Y. Sinha, and K. Haribabu, "A survey: Hybrid sdn," *Journal of Network and Computer Applications*, vol. 100, pp. 35 – 55, 2017.
- [15] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless Network Functions: Breaking the Tight Coupling of State and Processing," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, (Boston, MA), pp. 97–112, Mar. 2017.
- [16] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic Scaling of Stateful Network Functions," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, (Renton, WA), pp. 299–312, Apr. 2018.
- [17] T. Lin, B. Park, H. Bannazadeh, and A. Leon-Garcia, "SAVI Testbed Architecture and Federation," in *Future Access Enablers for Ubiquitous and Intelligent Infrastructures*, pp. 3–10, 2015.
- [18] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, Mar. 2008.
- [19] B. Park, T. Lin, H. Bannazadeh, and A. Leon-Garcia, "JANUS: Design of a Software-Defined Infrastructure Manager and its Network Control Architecture," in *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, pp. 93–97, June 2016.
- [20] "Transport Level Passive Ping Network Observer." [Online] Available: <http://www.pollere.net/pping.html> [Accessed: April 2019].