UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona

telecos BCN

Wide Area Network Autoscaling for Cloud Applications

Master Thesis
submitted to the Faculty of the
Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
by

Berta Serracanta Pujol

In partial fulfillment
of the requirements for the
Master's degree in Telecommunications **Engineering**

Advisor: Albert Cabellos Aparicio
Co-Advisor: Fabio Maino (Cisco)
Barcelona, June 2021

Acknowledgements

This master thesis would not have been possible without the help of many people. I would like to start thanking the UPC team, my advisor Albert Cabellos and Jordi Paillisse who have guided me through the whole process. Their collaboration has been invaluable.

Of course, I would also like to send my gratitude to my co-advisor Fabio Maino and all the Cisco team, Alberto Rodriguez-Nata, Lori Jakab and Elis Lulja, that have provided me not only with incredible technical challenges and discussions but also have made my internship very smooth in these strange virtual times.

And finally, thanks to my flatmates and friends, Mar i Marta for the game nights and laughs after hard days. Of course, I would also like to thank my family for their endless support. And last but not least to Tomàs, for always being there.

Abstract

Modern cloud orchestrators like Kubernetes provide a versatile and robust way to host applications at scale. One of their key features is autoscaling, that automatically adjusts cloud resources (compute, memory, storage) in order to dynamically adapt to the demands of the application. However, the scope of cloud autoscaling is limited to the datacenter hosting the cloud and it doesn't apply uniformly to the allocation of network resources. In I/O-constrained or data-in-motion use cases this can lead to severe performance degradation for the application. For example, when the load on a cloud service increases and the Wide Area Network (WAN) connecting the datacenter to the Internet becomes saturated, the application experiences an increase in delay and loss. In many cases this is dealt by overprovisioning network capacity, which introduces significant additional costs and inefficiencies.

On the other hand, thanks to the concept of "Network as Code", the WAN today exposes a programmable set of APIs that can be used to dynamically allocate and de-allocate capacity on-demand. In this thesis we propose extending the concept of cloud autoscaling into the network to address this limitation. This way, applications running in the cloud can communicate their networking requirements, like bandwidth or traffic profile, to an SDN controller or Network as a Service (NaaS) platform. Moreover, we aim to define the concepts of vertical and horizontal autoscaling applied to networking. We present a prototype that automatically allocates bandwidth in the underlay of an SD-WAN, according to the requirements of the applications hosted in Kubernetes. Finally, we discuss open research challenges.

Revision history and approval record

Revision	Date	Purpose
0	20/05/2021	Document creation
1	1/06/2021	Document revision
2	15/06/2021	Document revision

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Berta Serracanta Pujol	bertaserracanta@estudiantat.upc.edu
Albert Cabellos Aparicio	acabello@ac.upc.edu
Fabio Maino	fmaino@cisco.com

Written by:		Reviewed and approved by:	
Date	20/05/2021	Date	15/06/2021
Name	Berta Serracanta	Name	Albert Cabellos
Position	Project Author	Position	Project Supervisor

Contents

List of Figures

List of Tables

1 Introduction

During the last decade, Software-Defined Networking (SDN) has enabled programmability to network operation and management. By exposing expressive northbound APIs, the underlying physical network infrastructure can be controlled dynamically, offering unprecedented dynamic and quasi real time network control. Recently, several commercial networking scenarios have embraced the SDN paradigm, resulting in a new breed of networking solutions, such as SD-Wide Area Network [12, 13], or SD-Access Networks [11, 14].

Starting even earlier and driving the as-a-Service revolution, the *compute* environment has been following its own path of rapid innovation. Compute resources have become more granular, from physical hardware to VMs, then to containers [1, 2] and more recently towards Lambda functions [15, 16]. This makes it possible to consume compute resources more efficiently, as well as achieving the unprecedented scalability that has made Cloud Computing possible.

In this context, orchestration software to manage large pools of compute resources, such as OpenStack for virtual machines [17], or Kubernetes for containers [1] has become commonplace. These orchestration infrastructures have been naturally extended into the networking domain, especially in the case of service meshes, such as Envoy proxies controlled by Istio [23, 27]. However, the capability of the network to keep up with the scalability offered by the cloud infrastructure is limited mostly to datacenter networks (within the cluster). Inter-cluster connectivity, as well as the connection with the users, is typically provided via Wide-Area Networks (WAN), a relatively expensive resource with limited capacity.

In addition to the limited capacity of the WAN, the lack of communication between the cloud and the WAN hinders some of the advantages of the cloud. A key limitation emerges in the context of cloud autoscaling events. Autoscaling is a cloud feature to dynamically adapt the amount of compute and memory resources to the current application load, as a way to improve efficiency and optimize cost.

In current cloud deployments that span across datacenters and WANs, autoscaling events detected by the cloud orchestrator only propagate within the compute domain (i.,e, datacenter). Although an application can scale up to support higher loads, the connection towards the users or between clusters remains unchanged, which might cause congestion or reduce QoE. It may also require overprovisioning of network capacity, introducing significant additional costs and inefficiencies. Conversely, when applications scale down, network resources become idle and underutilized. Taking this into account, we argue that there is a need to develop the necessary abstractions and protocols to interface the *compute* and the *networking* domains, with a strong focus on the network autoscaling properties.

1.1 Objectives

In this thesis we explore how application-driven cloud autoscaling events can be proactively propagated to the network. Common approaches to network autoscaling are reactive, i.e. they monitor network traffic, compute an average, and trigger autoscaling events if the average exceeds a threshold [26]. However, in our case the objective was to make use of application context extracted from the cloud to proactively autoscale the network. This approach doesn't wait for surges in network load to increase capacity, but rather proactively increases network capacity shortly before surges in demand become observable in the network. With WAN autoscaling, the network dynamically matches the requirements of applications with the resources available in the network, achieving higher efficiency in resource utilization. Moreover, we intend to translate the cloud autoscaling concept to the network autoscaling environment. Also, another key objective was the developement of a proof of concept (PoC), its analisys and the identification of the future challenges that WAN autoscaling brings to the table.

1.2 Methods and procedures

This work is a continuation of the Cloud Native SD-WAN (CN-WAN) open source project developed by Cisco. This thesis is also leverages the cooperation between Cisco and PacketFabric, an underlay service provider by using the functionalities of their programmable API. Finally, this project also includes the work performed by Jordi Paillisse (UPC) on horizontal network autoscaling.

1.3 Work Plan

This thesis development has had a duration of a little over 6 months and the work plan followed for its elaboration has been structured in mainly four different work packages:

Research: this section is based on collecting information and related work on the technologies and topics related to this project. More specifically, four different topics have been studied. The first one was Software Defined Networking (SDN) applied to Wide Area Networks (WAN). The second one was studying the cloud computing world and its related technologies such as Kubernetes and Docker. The third one was centred around the CN-WAN project in order to get familiar with the source code as well as its concepts and components. Finally, the last section to study was the API provided by PacketFabric, which would be used for communicating the autoscaling events to the underlay.

Implementation: during this months a new version of the CN-WAN Adaptor component has been developed to allow the propagation of cloud autoscaling events to the WAN. Since developing a proof of concept is never linear, this section has been divided into the first complete prototype and the iterations that were necessary to develop the final one.

Testing: the next work package after completing the prototype consisted in data collection through different measures in order to analyse its performance. It also includes the discussion of this results and reaching the appropriate conclusions.

Documentation: the last work package consisted in documenting the code developed, a technical paper and this thesis.

1.4 Deviations and eventualities

The only major deviation of this project has been the reschedule of the technical paper deadline from the 21st of May 2021 to the 31st of May 2021.

1.4.1 Gantt Diagram

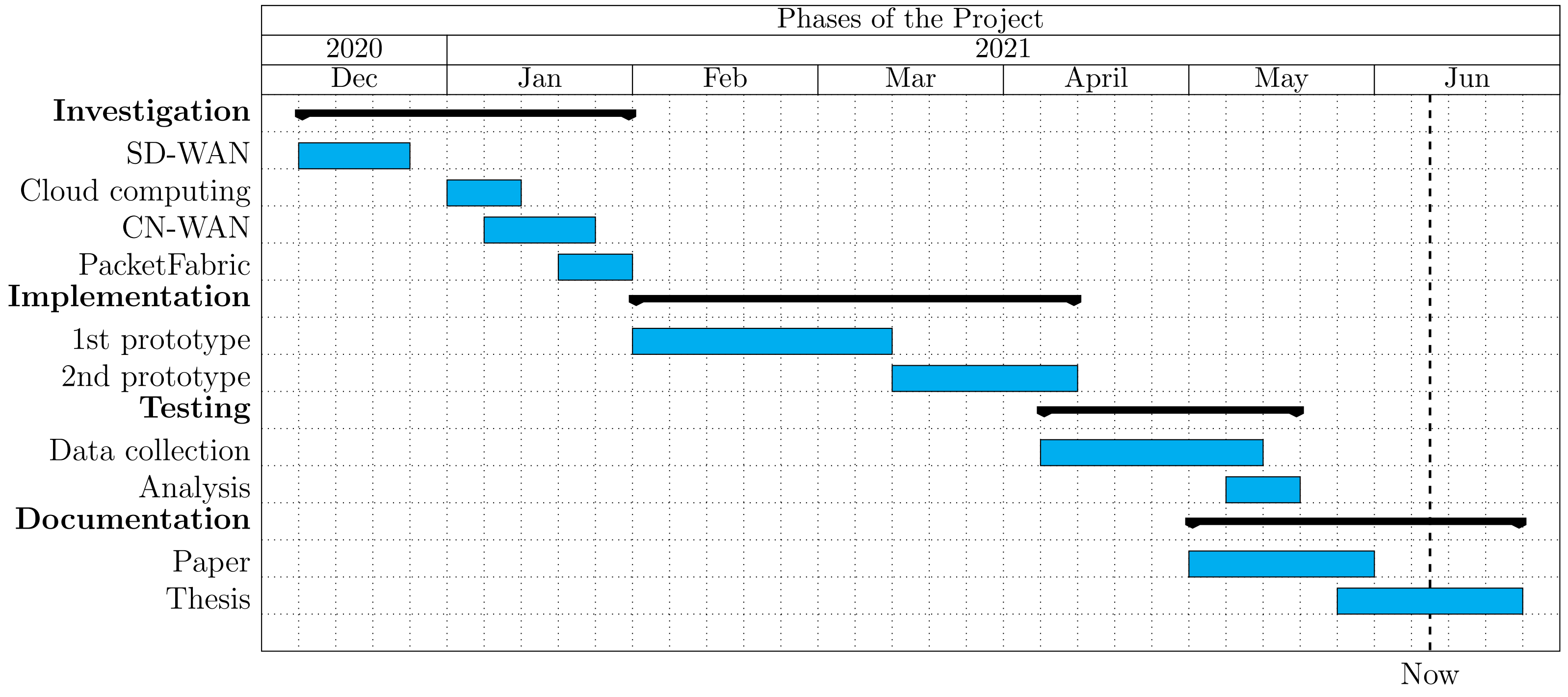


Figure 1: Gantt diagram of the project

2 State of the art

In this section we explore the background necessary for the project development, in particular the software defined networking and the hardware virtualisation. Moreover, a list of relevant literature is detailed, along with the similarities and differences with this project.

2.1 Software Defined Networks

Software Defined Networks (SDN) appeared around 2008 as a response to traditional networks limitations. Its static hardware and complex architecture led to scalability and troubleshooting issues in a cloud-centered world. With a strong focus on this, SDNs introduce network virtualization and the concept of abstracting the underlying network from the applications and services. SDNs are based on the dissociation of the control plane and the data plane, separating the routing and the forwarding processes on different devices. This decoupling enables features such as:

- The agility to dynamically adjust traffic flows to match its needs.
- The application of policies from a centralized management.
- The programmability of the network control, allowing to quickly configure, manage, optimize and automate the network resources.
- The simplification of the network design due to open standards.

SDNs have many applications, among them software-defined mobile networking, software-defined Local Area Networks. This particular thesis leverages the SDN concept applied to Wide Area Networks (SD-WAN). It is focused the optimization of resources cost reduction by leveraging different connection times on the underlay such as MPLS or 5G.

2.2 Hardware Virtualisation

In the past few decades dedicated hardware has transitioned into virtual machines, virtual environments that run isolated in a host machine with each own dedicated resources such as CPU, operating system and memory. More recently, containers have appeared as the next step of the transition. Based on operating system level virtualization, multiple isolated and secure virtualied containers can run on one physical device. Every container has all the necessary resources to run a specific application and in turn, the application perceives the container as a whole physical device.

With this paradigm appeared container orchestration, which consists in the automation in configuration, management, and coordination of containers. Kubernetes has become the de-facto orchestration tool. Besides of offering services as automate deployment or management, Kubernetes also provides autoscaling features. In particular, autoscaling can be divided into:

- Vertical Autoscaling: the controller automatically scales the resources, such as CPU or memory, on already deployed replicas

- Horizontal Autoscaling [4]: the controller automatically scales the number of replicas deployed based on configurable metric thresholds such as CPU utilisation.

2.3 Related work

We can find several works discussing network autoscaling, but they are usually limited to a specific domain, such as the data center [5], virtual network functions [6], or video streaming [3]. However, in our case: (i) we focus on the WAN, and (ii) take an architectural approach instead of an algorithmic one, because we make use of existing knowledge from the cloud to integrate it in the network policies.

Regarding networks that are aware of the application, a classical approach has been extending the socket API [21]. Contrarily, we do not modify the host stack but use API-driven interfaces to make the network aware of application requirements. Other works in a similar direction to ours focus on the connections between end users and ISPs [19], or inter-datacenter connections [20], while our proposal is centered on cloud applications and WANs. A more recent proposal suggests merging the L3 stack with the L7 stack (i.e. service meshes), in order to increase performance and application awareness in the network [22].

Other proposals in this field include Dawn [30], that annotates each application flow with direct network control, giving network control in the context of end hosts instead of the network controller. Our work also provides a good use case for ALTO protocol [31] implementations and future protocol extension work. ALTO is a protocol which with the intent to allow hosts to benefit from the network infrastructure by having access to a pair of maps: a topology map and a cost map.

Finally, this project lists further research challenges, some of which have already existing research lines such as the multiconnection optimization [32] or the study of the network resource allocation for each application [33].

3 Network Autoscaling

In this section we attempt to translate the term of autoscaling, vertical and horizontal, in the cloud into the network world.

Autoscaling is a cloud feature to make efficient use of compute and memory resources by automatically adapting their allocation to the current application load. This is done either by adding/removing more resources to an application instance (vertical autoscaling) or by adding/removing new application instances (horizontal autoscaling).

Therefore, we propose extending these two concepts to the networking domain in the following way (fig. 2): first, we make use of the API exposed by cloud orchestrators (eg. Kubernetes) to determine the network needs of the applications running in the cloud, as determined by application-level context available in the cloud. For example, we can detect cloud autoscaling events that indicate that the demand for a given application is increasing and may require more network capacity.

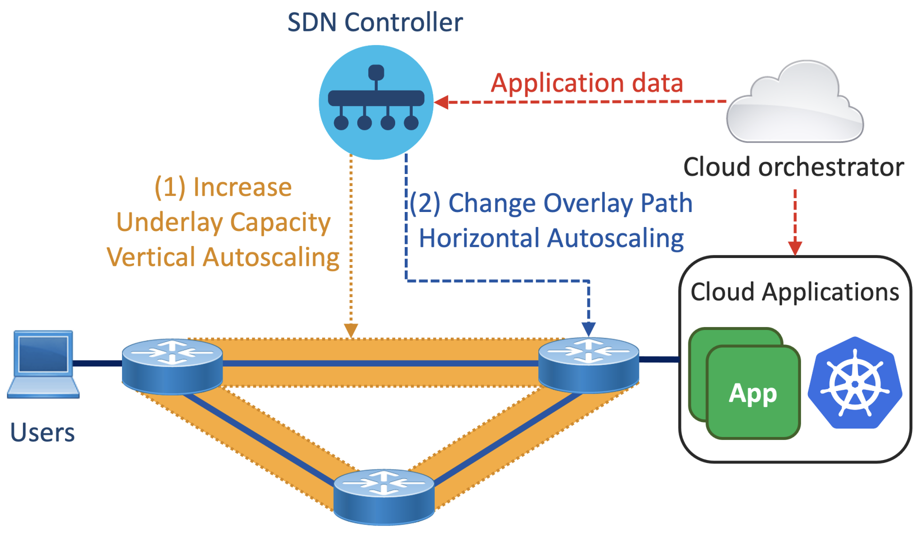


Figure 2: Vertical and Horizontal Network Autoscaling

Then, we send this information to an SD-WAN controller so it can be programmed by NetOps to act upon the network. For example, we can request more bandwidth over a specific connection to an underlay provider (1), or change some paths according to the requested traffic profile (2).

Note that we aim to clearly separate the jobs of DevOps and NetOps with a simple interface. On one hand, DevOps make use of cloud tools to add context information about the application. In our case, they add annotations to define the traffic profile needed by a given application (annotations are typically fairly abstract indications about the application network behavior using labels such as “file transfer”, “video streaming”, or ”database analytics”). On the other hand, NetOps can automate the optimal allocation of network resources to applications thanks to their in-depth knowledge of the WAN infrastructure. In the next sections, we describe in detail the concepts of vertical and horizontal network autoscaling.

3.1 Vertical Network Autoscaling

In the cloud, vertical autoscaling modifies the properties of an existing compute instance. Similarly, we define Vertical Network Autoscaling as changing the properties of an existing connection (e.g, L2 pseudo-wires, MPLS tunnel, VXLAN [24] or LISP tunnels [25], etc). For example, if the number of instances of a specific application increases, we will increase accordingly the bandwidth of the connection serving these applications.

We are assuming that: (i) the resource limiting application performance is the amount of WAN bandwidth, and (ii) the SD-WAN controller has access to an API-driven underlay provider that can dynamically provision the capacity of the connections, as it is becoming common with NaaS providers [27].

3.2 Horizontal Network Autoscaling

Taking into account that cloud horizontal autoscaling means adding more instances to handle growing demand, we define Horizontal Network Autoscaling as changing the path inside the network that a specific application is currently following, due to changes in its requirements. For instance, consider that a video streaming application can tolerate latency up to a certain maximum, and that DevOps have labeled such application in the cloud orchestrator. We can use these labels to pull information from the orchestrator to identify this application in the network, e.g. via IP and port. Since most SD-WAN controllers monitor path properties like delay, jitter or bandwidth, we can use this information to steer application flows through the appropriate paths in the SDN network via traffic engineering or Segment Routing (SR).

However, note that when we change the path inside the network we do not necessarily mean inside the *same* network. In other words, we can dynamically choose from different providers to adapt better to sudden increases in load, using resources more efficiently. This is especially relevant from a business perspective, because increasing efficiency translates to reductions in cost through price arbitrage. For example, consider a virtual circuit of 1 Gbps that can be extended up to 2 Gbps, but the price of this additional Gbps is significantly higher than the first. Upon a surge in application traffic, instead of contracting this additional Gbps that is more expensive, we can request it to to a different provider that offers a less expensive alternative. More details about this can be found in section 6.

4 Implementation

In this section we detail the procedure of implementing the studied scenarios, its development and we analyse the obtained results. First, in the vertical network autoscaling scenario and finally in the horizontal network autoscaling case.

4.1 Vertical Network Autoscaling

4.1.1 Scenario

The experimental setup consisted of three main parts: cloud, network, and cloud-network API (fig. 3).

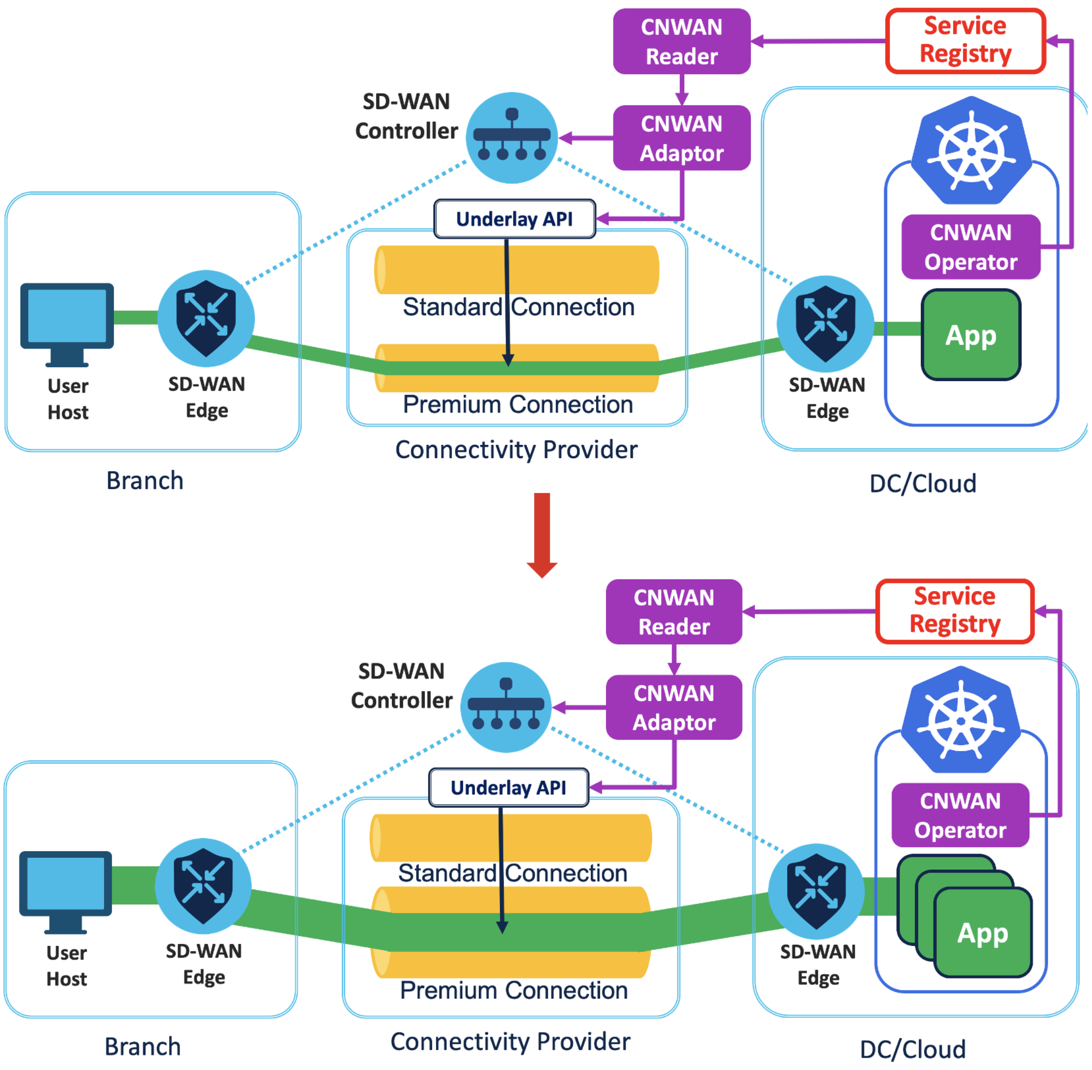


Figure 3: Vertical scenario overview

Cloud: We used a public cloud for our experiments [8] controlled by Kubernetes (Version 1.18.16-gke.502), and an ad-hoc HTTP Echo server [9] as a cloud application.

We leveraged the functionalities of the Kubernetes Horizontal Pod Autoscaler (HPA) to generate autoscaling events. The HPA is based on a simple algorithm that operates between a target metric and the current value of the metric, adding or deleting replicas if the current state does not match the desired state. In our implementation, the target metric was the average CPU utilization across all monitored pods. We set the HPA threshold to 40%, meaning that when CPU utilization is greater than 40%, Kubernetes deploys new containers, scaling from 1 up to 150 replicas.

We used a custom HTTP Loader to increase the load on the HTTP Echo server in the Kubernetes containers. The HTTP Loader artificially generated 700 HTTP echo requests, and we increased this number to 1900, 2900, 3500, and 4500 connections, at 30, 90, 150, 210 seconds since the start of the test, respectively.

Network: we used a commercial underlay network connecting Washington D.C. and Seattle based on a Segment Routing tunnel [10], with a baseline of 50 Mbps and a maximum capacity of 100 Gbps. This network is programmatically controlled by a proprietary API [27]. Fundamentally, the API allows to create, upgrade and delete virtual circuits between the two established locations. This network connected the client (HTTP Loader in Washington D.C.) with the cloud application (HTTP Echo server in Seattle).

Cloud-Network API: we developed an open-source API that allows the network and the cloud to talk, the Cloud-Native SD-WAN project (CN-WAN [7]). The main role of CN-WAN is to provide an interface between cloud applications and networks that connect to end-users, or applications running across multiple clusters. This interface allows us to: (i) identify which cloud applications require network autoscaling, (ii) communicate autoscaling events from the cloud to the network, and (iii) quantify the scaling factor required on the underlay. CN-WAN consists of three main blocks: the Operator, the Reader and the Adaptor (fig. 6). When Kubernetes detects a variation in the number of replicas of the application being monitored, the Operator publishes this change on an external service registry. From there, the Reader polls the service registry and announces the events to the Adaptor, which in turn contacts the programmable underlay to adjust the network to the upcoming bandwidth requirements.

4.1.2 Development

The Vertical Network Autoscaling solution developed leverages existing functionalities of the CN-WAN project and PacketFabric. In order to implement it the CN-WAN Oerator has been modified to report the changes in the number of replicas of a targetted cloud application. In the event of new replicas deployed or existing replicas deleted, the Operator publishes the new amount of pods to the external service registry. The CN-WAN Reader now pulls the service registry to detect any update and it makes an API call to a new CN-WAN Adaptor to make the according changes in the underlay.

The CN-WAN adaptor has been developed following these assumptions: (i) there are no third party modifications on the virtual circuit capacity, (ii) it fits into a point-to-point scenario and (iii) the virtual circuit is created with a baseline bandwidth that cannot be removed.

For the implementation, this component has been developed as a Docker container, for portability and flexibility. The API that connects the CN-WAN Reader with the PacketFabric underlay has been implemented using the OpenAPI specification [28] using a Flask server [29] and written in Python programming language.

This new Adaptor has three different main functionalities. The first one is to create a mapping between each replica and the network load assigned to it, in Mbps (*bw_per_replica* in eq. 1 and 2). The second functionality takes place on the reception of a cloud autoscaling event from the Reader (*old_replicas* in 1). The Adaptor needs to keep the state of the existing number of pods deployed for each targeted application (*old_replicas* in 2) in order to quantify the variation in the autoscaling event. With the variation of replicas accounted for, then the computation of the required underlay bandwidth takes place. To do this computation it is also important to take into account that the virtual circuit baseline bandwidth and the bandwidth consumed by other applications that autoscale. With these two values the new required bandwidth can be calculated as:

$$bw = old\_bw - old\_replicas \times bw\_per\_replica \tag{1}$$

$$new\_bw = bw + new\_replicas \times bw\_per\_replica \tag{2}$$

The total new bandwidth required to fit the demand of all the replicas deployed then needs to be modified to adjust to the granularity provided by the underlay.

Finally if the final bandwidth computed is different from the one currently allocated on the virtual circuit, the Adaptor generates an API call to PacketFabric to request an update on the virtual circuit.

Besides the main functionalities, other features such have been implemented such as the ones listed below. A picture of all the API endpoints implemented can be found in appendix C.

- Log in with PacketFabric credentials.
- API endpoints to create/delete/reset new virtual circuits.
- API endpoint to modify the bandwidth per replica assigned to a targeted cloud application.
- API endpoints to retrieve stored information such as current number of replicas, bandwidth allocated on the virtual circuit, bandwidth allocated for a specific application, between others.
- HTTP mock server that acts as the underlay for testing purposes.

The CN-WAN Adaptor source code will be open sourced when it is no longer under review.

4.1.3 Results

Fig. 4 shows an overview of the operation of vertical network autoscaling. We have measured the throughput in the cloud router connected to the virtual circuit, the total capacity of the connection, and the number of application replicas in Kubernetes. The plot shows how the bandwidth allocated in the underlay's virtual circuit starts autoscaling proactively with the application to support upcoming traffic.

More specifically, we started injecting traffic to a 50 Mbps virtual circuit to generate autoscaling events. We can see that traffic starts increasing around 50 seconds after the experiment starts (blue line), that simulates a spike in demand for a particular cloud application. In parallel, as traffic increases, the CPU utilization of the cloud application rises, and Kubernetes automatically deploys more replicas (orange line, at the same time as traffic increases). We can see that shortly after the number of replicas is greater than 50, the underlay capacity is increased from 100 Mbps to 200 Mbps (at around 140 seconds in the timescale). This is because we trigger autoscaling events when traffic is greater than the minimum allocated bandwidth in the virtual circuit (50 Mbps). We estimate traffic assuming that each replica consumes a fixed amount of bandwidth, 1 Mbps in this case. Since at this moment there are aprox. 75 replicas, this translates to 75 Mbps, so we jump to the next bandwidth step allowed by the underlay provider (here from 100 to 200 Mbps).

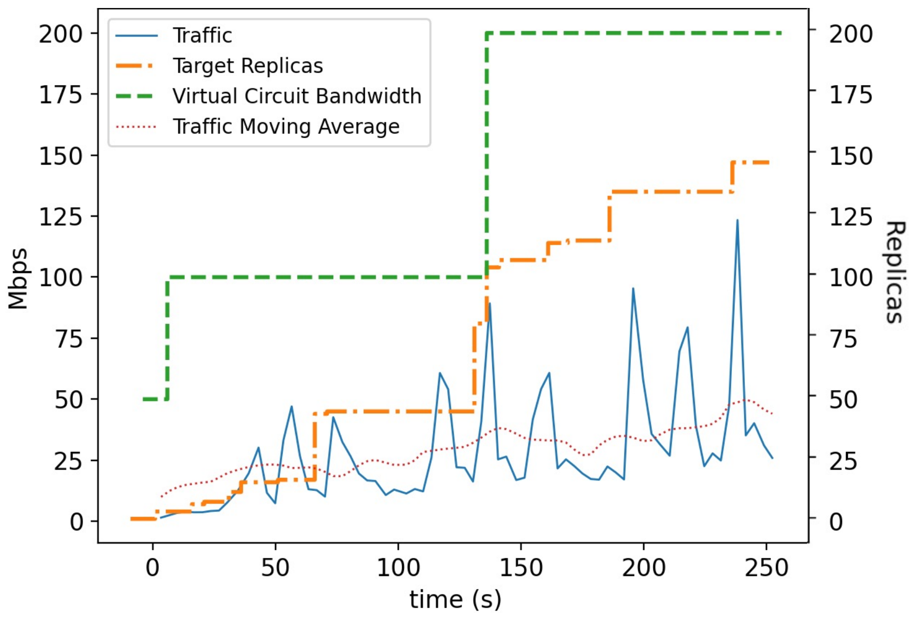
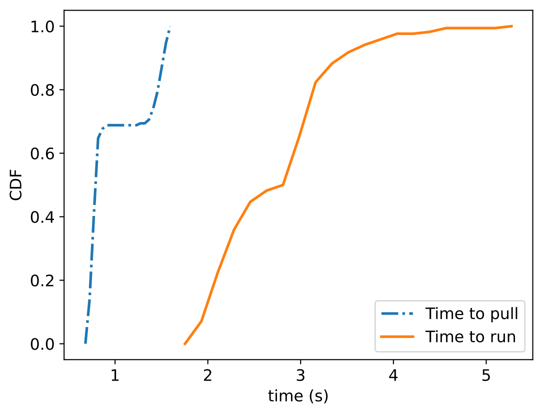


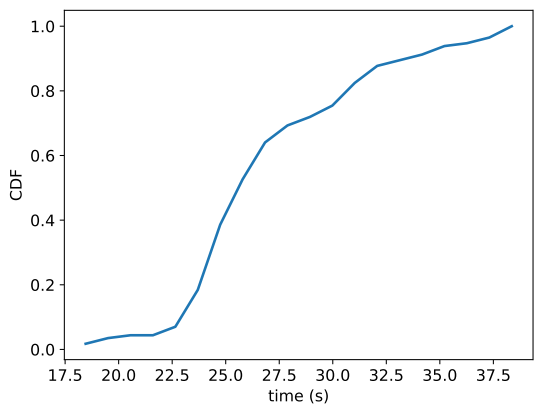
Figure 4: Vertical autoscaling performance

We must remark that the increase in virtual circuit capacity is triggered proactively, as soon as the cloud application starts autoscaling (the sudden increase of 50 to 75 replicas is very close in time to the increase of virtual circuit bandwidth). This means that the underlay virtual circuit capacity is increased *before* the actual network traffic hits the 100 Mbps initial capacity of the underlay, i.e, the blue line never crosses the green one.

Furthermore, as soon as the cloud application scales down because of diminished demand, the network will scale down the underlay capacity accordingly (not shown in the graph), resulting in a highly efficient and cost effective usage of the underlay resources. This example shows the benefits of autoscaling the WAN, compared with today's typical approach of overprovisioning underlay capacity to address peaks of demand, or common autoscaling techniques that leverage traffic prediction models based on autoregression or moving average to decide when to scale up or down [26].



(a)



(b)

Figure 5: Kubernetes controller (a) and overlay controller (b) response time

Next, we focus on the response time of Kubernetes and the overlay controller when reacting to autoscaling events. The plot in fig. 5a shows the Cumulative Distribution Function (CDF) of the Kubernetes Horizontal Pod Autoscaler (HPA) latency in two situations. *Time to pull* corresponds to the delay to pull the container image when we receive a new autoscaling event.

Time to run is the delay between the trigger of an autoscaling event by the HPA until Kubernetes indicates that the new container is running. Note that this measurement includes the time to pull the image, and that each CDF contains 170 measurements. In both cases, we can see that Kubernetes can autoscale in the order of seconds.

Finally, the plot in fig. 5b shows the CDF of the delay of the SD-WAN overlay controller, i.e., the time between receiving an autoscaling event in the overlay until the capacity of the virtual circuit is updated. We can see that this time is in the order of tens of seconds, one order of magnitude more than the Kubernetes controller response time.

Other results, such as a submission to the Network-Application Integration SIGCOMM Workshop (2021) and the presentation during KubeConEU ScaleX event (2021) can be found in appendices A and B, respectively.

4.2 Horizontal Network Autoscaling

4.2.1 Scenario

We modified the setup of section 4.1 in order to steer a traffic flow between two different underlay tunnels. More specifically: (i) instead of the HTTP echo server, we deployed a container that streamed video from the Kubernetes cluster, (ii) we connected a VM to receive the video stream on the other end of the WAN, (iii) we annotated the Kubernetes container with a label to identify it as a video stream, and (iv) we configured the SD-WAN controller with two tunnels: one with a limited bandwidth of 3 Mbps, an another with 1 Gbps.

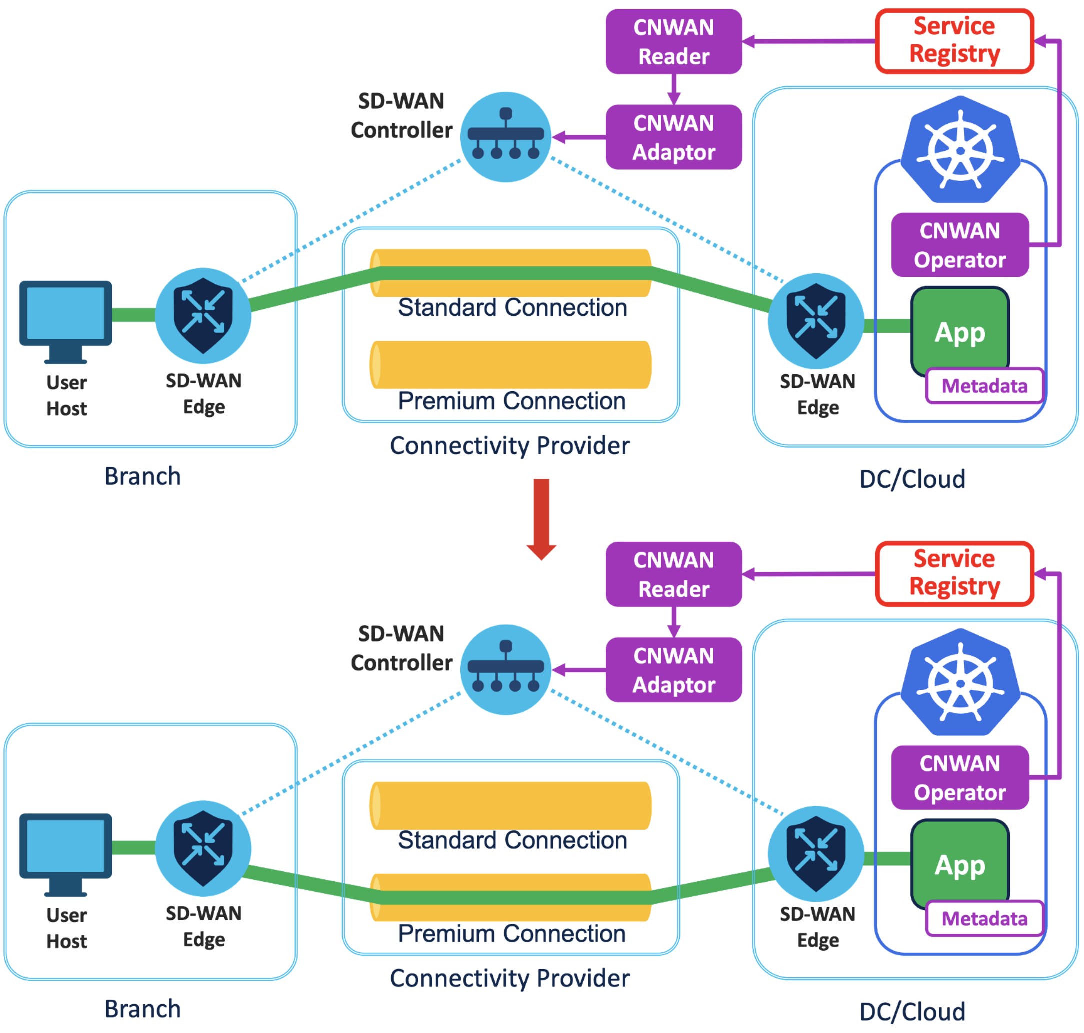


Figure 6: Horizontal scenario overview

This way, the CN-WAN operator reads the IP and port of the video stream container from Kubernetes and stores it in a service registry. Then, the reader collects this information and sends it to the SD-WAN controller via the adaptor, indicating the IP address and port of the Kubernetes application, and which tunnel it should take.

In order to change the path of the video stream, we modified several times the annotation in Kubernetes, so the flow alternated between the two tunnels.

4.2.2 Results

We can see an example of the video stream switching paths operation in fig. 7, that shows the bandwidth in both tunnels. We can appreciate that while one tunnel has traffic, the other doesn't, and that the throughput never exceeds 3 Mbps in the rate-limited tunnel.

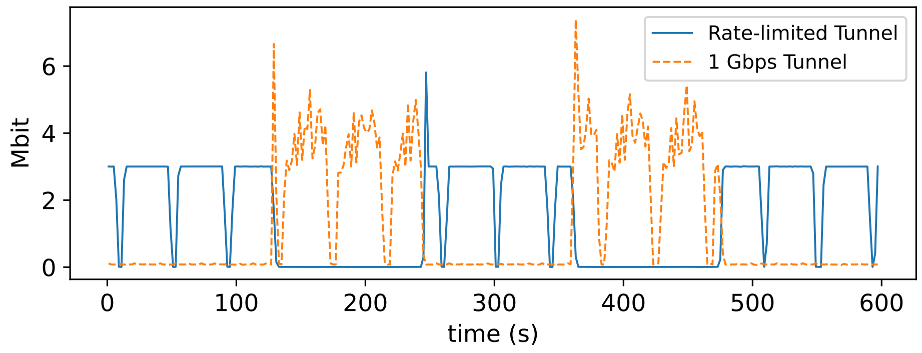


Figure 7: Throughput in the rate-limited tunnel and the 1 Gbps tunnel

Finally, we measured the delay between changing the application annotation in Kubernetes until traffic appeared in the other tunnel. We considered that traffic appeared in the other tunnel when it received more than 1 Mbps. We repeated the experiment 200 times, and the delay was below 23 seconds 80% of the times, with a maximum of 31 seconds.

5 Budget

This section details the budget dedicated to the development of this project. Since it is a software-based project, the more significant part of the budget consists on the resources dedicated to human resources. To account for them, table 1 and 2 show the amount of hours dedicated by the personnel. The work of the engineered has been computed at 10 €/hour and the work of the managers and senior engineers at 20 €/hour. Spanish social services taxes have also been considered.

Table 1: Budget dedicated to the engineer in charge of the project

Task	Hours	Retribution/h	Social Services (30%)	Total (€)
Research	200	10	600	2600
Implementation	120	10	200	5200
Documentation	400	10	360	1560
Total:				9360 €

Table 2: Budget dedicated to the manager in charge of the project supervision

Task	Hours	Retribution/h	Social Services (30%)	Total (€)
Support	50	20	300	1300
Meetings	48	20	288	1248
Total:				1548 €

In table 3, the budged dedicated to equipment has been computed. For that we have considered the use of a 13-inch MacBook Pro with a 2,4 GHz Quad-Core Intel Core i5 processor and 16 GB 2133 MHz LPDDR3 memory. To compute the amortisable value of the computer, a 10% residual value has been taken into account. Virtual machines used for the deployment of the testbed have also been accounted for. A total of 6 Linux e2-medium (2 vCPUs, 4 GB memory) VMs have been deployed in Google Cloud Platform.

Table 3: Budget dedicated to the equipment used during the project time spant

	Units	Cost	Amortisation	Useful life	Total (€)
Computer	1	2500	2250	4	281,25
Linux Machines	6	450/month	-	-	2700
Total:					2981.25 €

Finally, the last resources that need accounting are the software licenses. For programming, the free UI Visual Studio Code has been used. For code sharing and control version we have used Github since it has a free license. For testing and developing the APIs, Postman provided all the tools needed without fees. Finally, the Viptela SDN software has been considered free for the development of this project since it has been developed in cooperation with Cisco, the vendor.

Table 4: Budget dedicated to the licenses required for the project development

License	**Cost (€)**
Visual Studio Code	Free
Postman	Free
Github	Free
Viptela SDN	Free
Total:	Free

After adding all the expenses the final project budget adds up to a total of 13889,25 €.

6 Conclusions and open research challenges

In this last section we detail the conclusions extracted during the development of the project and we finish by presenting future research challenges that we have identified.

6.1 Conclusions

The softwarization of the WAN and the advent of connectivity service providers that are exposing APIs to dynamically provision underlay capacity [27], together with the development of an application-to-network API makes proactive autoscaling of WAN resources as easy as autoscaling compute and memory. This is a very significant improvement, compared with the status quo of overprovisioning WAN capacity to address peaks of demand, as well as from solutions that monitor network utilization reacting to congestion and saturation. The basic prototype of WAN autoscaling that we implemented shows that it is possible to extend the scope of autoscaling to the WAN.

6.2 Open research challenges

To conclude this thesis we list a number of future research challenges that are worth further researching.

Bandwidth Estimation: regarding vertical network autoscaling, it is crucial to perform an accurate estimation of the bandwidth required by each compute replica, in order to maximize efficiency, prevent congestion or avoid overprovisioning bandwidth, and its associated cost. While in our prototype we let the developers configure this parameter, a more accurate approach is to automatically estimate it using sophisticated techniques. There is extensive research on bandwidth estimation [18], with monitoring tools or prediction algorithms, either linear or non-linear prediction models. Additional research is needed to understand how such existing techniques can benefit from having information from the cloud orchestrator.

Bandwidth Granularity: the bandwidth allocation granularity offered by the overlay controller is key to avoid overprovisioning. That is, the overlay controller can increase or decrease the allocated bandwidth in certain steps (e.g, 10, 50, 100 Mbps). Larger granularities that can match the required bandwidth of the compute replicas will lead to very efficient use of the resources. However, offering large granularities (e.g., steps of 1 Mbps) complicates the SDN overlay controller management. In addition, very large granularities (e.g, 1 kbps) can lead to issues handling traffic bursts. As a result, additional research is required in this field.

Provisioning Time: in the context of underlay providers, the maximum capacity of a tunnel is typically a management plane configuration parameter. In many network scenarios, the provisioning time (i.e., the time until the setting takes effect) is not considered relevant and has been often done manually. In recent years programmability has allowed this time to be significantly faster, changing from manual (e.g, through phone calls) to tenths of seconds [27]. Despite this huge improvement, in our scenario this provisioning delay would need to commensurate with the reaction time of the cloud autoscaling engine (currently few seconds), in order to avoid under-utilization or service disruption. Additional research efforts are required to address this challenge.

Application Diversity: in our experiments, we have inherently assumed that each application is assigned to a single tunnel. However, in real-world scenarios, different applications can share the same connection. In order to address this, the overlay control plane should take into account that multiple applications share the same physical link, and scale capacity accordingly.

Network Topology: our solution is designed for scenarios with point-to-point connections, where all the end-users are connected through a single tunnel to the datacenter running the application. With this setup, autoscaling events for an application only affect one tunnel. However, in real-world scenarios, a datacenter servicing end-users can be attached to different access networks and thus, each access network uses a different tunnel towards the datacenter. In this case, upon a surge in application traffic, and from the perspective of the cloud orchestrator, we don't know which virtual circuit needs additional bandwidth. Hence, we need more information about the traffic to determine which connections must be scaled. For example, we could use traffic monitoring techniques, or take a more proactive approach by tracking the incoming connections to the cloud datacenter.

Billing Model: WAN autoscaling affords a new consumption model for WAN capacity where bandwidth is provisioned (and de-provisioned) on demand in quasi real time, rather than through overprovisioning. The billing model offered by today's underlay connectivity providers reflects the overprovisioning consumption model offering increasing SLAs (such as 1, 10, 50 Gbps) typically billed on a monthly basis. WAN autoscaling may drive the introduction of a new billing model where WAN consumption is billed on demand. We believe this may lead to a more efficient use of network infrastructure resources for the service provider, that will in turn be reflected to a lower cost for the end user. Future work in this area should include a detailed analysis of billing models for WAN autoscaling, and their comparison with current billing models.

References

[1] D. Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[2] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172, 2015.

[3] D. Niu and Hong Xu et al. Quality-assured cloud bandwidth auto-scaling for video-on-demand applications. In *2012 Proceedings IEEE INFOCOM*, pages 460–468, 2012.

[4] The Kubernetes Project. Horizontal pod autoscaler, 2021.

[5] Hitesh Ballani and Paolo Costa et al. Towards predictable datacenter networks. *SIGCOMM Comput. Commun. Rev.*, 41(4):242–253, August 2011.

[6] S. Rahman et al. Auto-scaling vnfs using machine learning to improve qos and reduce cost. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–6, 2018.

[7] Fabio Maino, Alberto Rodriguez-Natal, Lori Jakab, and Elis Lulja. Cloud-native SD-WAN (CN-WAN), 2021. Available at `https://github.com/CloudNativeSDWAN/cnwan-docs`.

[8] Google. Google cloud platform, 2021. Available at `https://cloud.google.com/`.

[9] Elis Lulja. Echo-server, 2021. Available at `https://github.com/SunSince90/echo-server`.

[10] Clarence Filsfils, Stefano Previdi, Les Ginsberg, Bruno Decraene, Stephane Litkowski, and Rob Shakir. Segment Routing Architecture. RFC 8402, July 2018.

[11] Jordi Paillisse and Marc Portoles et al. Sd-access: Practical experiences in designing and deploying software defined enterprise networks. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '20, page 496–508, New York, NY, USA, 2020. Association for Computing Machinery.

[12] Sushant Jain and Alok Kumar et al. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 3–14, New York, NY, USA, 2013. Association for Computing Machinery.

[13] Xin Jin, Yiran Li, Da Wei, Siming Li, Jie Gao, Lei Xu, Guangzhi Li, Wei Xu, and Jennifer Rexford. Optimizing bulk transfers with software-defined optical wan. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 87–100, New York, NY, USA, 2016. Association for Computing Machinery.

[14] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. *SIGCOMM Comput. Commun. Rev.*, 37(4):1–12, August 2007.

[15] Adam Eivy and Joe Weinman. Be wary of the economics of ”serverless” cloud computing. *IEEE Cloud Computing*, 4(2):6–12, 2017.

[16] Maciej Malawski, Kamil Figiela, Adam Gajek, and Adam Zima. Benchmarking heterogeneous cloud functions. In Dora B. Heras, Luc Bougé, Gabriele Mencagli, Emmanuel Jeannot, Rizos Sakellariou, Rosa M. Badia, Jorge G. Barbosa, Laura Ricci, Stephen L. Scott, Stefan Lankes, and Josef Weidendorfer, editors, *Euro-Par 2017: Parallel Processing Workshops*, pages 415–426, Cham, 2018. Springer International Publishing.

[17] Tiago Rosado and Jorge Bernardino. An overview of openstack architecture. In *Proceedings of the 18th International Database Engineering & Applications Symposium*, IDEAS ’14, page 366–367, New York, NY, USA, 2014. Association for Computing Machinery.

[18] R. Prasad, C. Dovrolis, M. Murray, and K. Claffy. Bandwidth estimation: metrics, measurement techniques, and tools. *IEEE Network*, 17(6):27–35, 2003.

[19] M. El-Gendy, A. Bose, S.-T. Park, and K.G. Shin. Paving the first mile for qos-dependent applications and appliances. In *Twelfth IEEE International Workshop on Quality of Service, 2004. IWQOS 2004.*, pages 245–254, 2004.

[20] Felipe Rodriguez Yaguache. Enabling Edge Computing Using Container Orchestration and Software Defined Networking. Master’s thesis, Aalto University. School of Electrical Engineering, 2019.

[21] Philipp S. Schmidt et al. Socket intents: Leveraging application awareness for multi-access connectivity. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT ’13, page 295–300, New York, NY, USA, 2013. Association for Computing Machinery.

[22] Gianni Antichi and Gábor Rétvári. Full-stack sdn: The next big challenge? In *Proceedings of the Symposium on SDN Research*, SOSR ’20, page 48–54, New York, NY, USA, 2020. Association for Computing Machinery.

[23] Wubin Li and Yves Lemieux et al. Service mesh: Challenges, state of the art, and future research opportunities. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 122–1225, 2019.

[24] Mallik Mahalingam, Dinesh Dutt, Kenneth Duda, Puneet Agarwal, Larry Kreeger, T. Sridhar, Mike Bursell, and Chris Wright. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. RFC 7348, August 2014.

[25] Dino Farinacci, Vince Fuller, David Meyer, and Darrel Lewis. The Locator/ID Separation Protocol (LISP). RFC 6830, January 2013.

[26] Multi-scale high-speed network traffic prediction using k-factor gegenbauer arma model. In *2004 IEEE International Conference on Communications (IEEE Cat. No.04CH37577)*, volume 4, pages 2148–2152 Vol.4, 2004.

[27] What is naas?, 2021. Available at `https://packetfabric.com/blog/what-is-naas`.

[28] Openapi specification, 2021. Available at `https://spec.openapis.org/oas/v3.1.0`.

[29] Flask documentation, 2021. Available at `https://flask.palletsprojects.com/en/2.0.x/`.

[30] Shuhe Wang, Dong Guo, Wei Jiang, Haizhou Du, and Mingwei Xu. Dawn: Co-programming distributed applications with network control. In *Proceedings of the Workshop on Network Application Integration/CoDesign*, NAI '20, page 14–19, New York, NY, USA, 2020. Association for Computing Machinery.

[31] Shu Yang, Laizhong Cui, Mingwei Xu, Y. Richard Yang, and Rui Huang. Delivering Functions over Networks: Traffic and Performance Optimization for Edge Computing using ALTO. Internet-Draft draft-yang-alto-deliver-functions-over-networks-01, Internet Engineering Task Force, July 2020. Work in Progress.

[32] Hongqiang Harry Liu, Ye Wang, Yang Richard Yang, Hao Wang, and Chen Tian. Optimizing cost and performance for content multihoming. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, page 371–382, New York, NY, USA, 2012. Association for Computing Machinery.

[33] Hao Yin, Chuang Lin, Berton Sebastien, Bo Li, and Geyong Min. Network traffic prediction based on a new time series model. *International Journal of Communication Systems*, 18(8):711–729, 2005.

Appendices

A NAI SIGCOMM Workshop submission

SIGCOMM is the flagship annual conference of the ACM Special Interest Group on Data Communication (SIGCOMM) on the applications, technologies, architectures, and protocols for computer communication.

The Network-Application Integration workshop searches for contributions to the design principles and real implementations of systems that enable network-application co-design. It focuses on realistic NAI designs, implementations and experiences, and exploring both sides of NAI: application-aware networking (AAN) and network-aware application (NAA).

This project has submitted a paper to the NAI SIGCOMM Workshop. If accepted it will be presented in August 2021.

B KubeCon EU 2021 ScaleX

Modern cloud native development can be complicated with the lack of true pipeline integration across tools for application platforms, security, and networking for an app-first world. ScaleX, intends to explore building for scalability and reliability and what that means for the modern cloud native developer.

This project was presented during this event by David Ward, the CEO of PacketFabric and Anna Claiborne, founder and VP of engineering.

Figure 8 shows a screenshot of the presentation. For viewer context, on the top right of the screen, there is Lens, the UI that monitors Kubernetes resources, it is tracking the Horizontal Pod Autoscaler. The autoscaler is set to trigger an autoscale event (which consists in adding more replicas) once the existing replicas reach 40% of their CPU utilisation. The replica column shows the number of current active replicas.

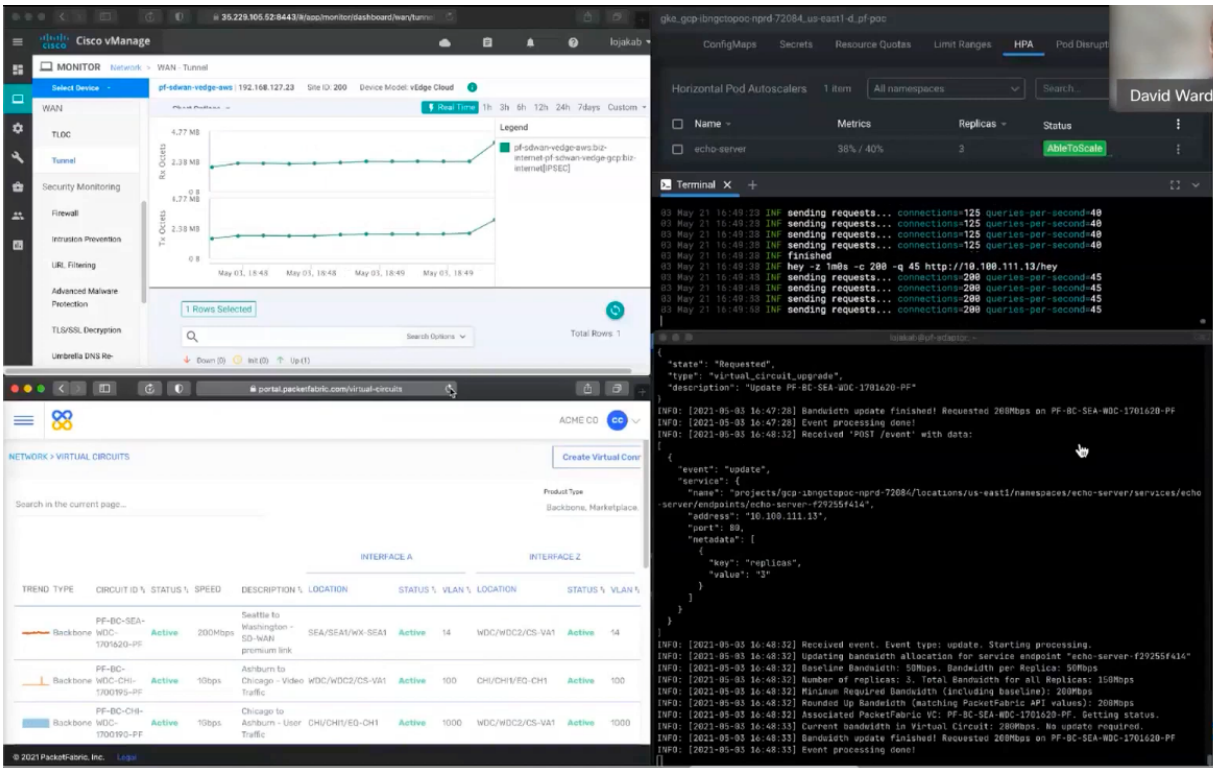
The application deployed is a simple echo server that processes HTTP requests. In the middle right there is the Linux client that generates the requests for the echo server. During the demonstration, sets of connections are being started periodically to generate increasing traffic over time.

On the bottom right, the logs of the CN-WAN Adaptor are shown. It displays how the autoscaling events are received and how these are translated to a call to the PacketFabric API after computing the amount of bandwidth needed to fit the demand.

On the bottom left, moving clockwise, there is the PacketFabric portal. On the first line it shows the virtual circuit connecting Seattle with Washington D.C. The browser auto refreshes to show how this is being updated with the autoscaling events.

Finally, on the top left there is Cisco vManage monitoring the traffic going over the SD-WAN tunnel over the PacketFabric virtual circuit. It shows the bandwidth consumed and how it increases over time. Note that the bandwidth actually consumed is much lower than the 50Mbps defined per replica, it is just a way of showing WAN autoscaling events with a simple echo server. This can be configured to match more properly the actual network load that each replica has.

The full presentation can be found at `https://www.youtube.com/watch?v=DEepmB7aWE0`.



Name	Metrics	Replicas	Status
echo-server	38% / 40%	3	AbleToScale

```
03 May 21 16:49:23 INF sending requests... connections=125 queries-per-second=40
03 May 21 16:49:23 INF sending requests... connections=125 queries-per-second=40
03 May 21 16:49:38 INF sending requests... connections=125 queries-per-second=40
03 May 21 16:49:38 INF sending requests... connections=125 queries-per-second=40
03 May 21 16:49:38 INF finished
03 May 21 16:49:38 INF hey -z 1m0s -c 200 -q 45 http://10.100.111.13/hey
03 May 21 16:49:48 INF sending requests... connections=200 queries-per-second=45
03 May 21 16:49:48 INF sending requests... connections=200 queries-per-second=45
03 May 21 16:49:53 INF sending requests... connections=200 queries-per-second=45
03 May 21 16:49:58 INF sending requests... connections=200 queries-per-second=45
```

```
{
  "state": "Requested",
  "type": "virtual_circuit_upgrade",
  "description": "Update PF-BC-SEA-WDC-1701620-PF"
}
INFO: [2021-05-03 16:47:28] Bandwidth update finished! Requested 200Mbps on PF-BC-SEA-WDC-1701620-PF
INFO: [2021-05-03 16:47:28] Event processing done!
INFO: [2021-05-03 16:48:32] Received 'POST /event' with data:
[
  {
    "event": "update",
    "service": {
      "name": "projects/gcp-ibngctopoc-nprd-72084/locations/us-east1/namespaces/echo-server/services/echo
-server/endpoints/echo-server-f29255f414",
      "address": "10.100.111.13",
      "port": 80,
      "metadata": [
        {
          "key": "replicas",
          "value": "3"
        }
      ]
    }
  }
]
INFO: [2021-05-03 16:48:32] Received event. Event type: update. Starting processing.
INFO: [2021-05-03 16:48:32] Updating bandwidth allocation for service endpoint "echo-server-f29255f414"
INFO: [2021-05-03 16:48:32] Baseline Bandwidth: 50Mbps. Bandwidth per Replica: 50Mbps
INFO: [2021-05-03 16:48:32] Number of replicas: 3. Total Bandwidth for all Replicas: 150Mbps
INFO: [2021-05-03 16:48:32] Minimum Required Bandwidth (including baseline): 200Mbps
INFO: [2021-05-03 16:48:32] Rounded Up Bandwidth (matching PacketFabric API values): 200Mbps
INFO: [2021-05-03 16:48:32] Associated PacketFabric VC: PF-BC-SEA-WDC-1701620-PF. Getting status.
INFO: [2021-05-03 16:48:33] Current bandwidth in Virtual Circuit: 200Mbps. No update required.
INFO: [2021-05-03 16:48:33] Bandwidth update finished! Requested 200Mbps on PF-BC-SEA-WDC-1701620-PF
INFO: [2021-05-03 16:48:33] Event processing done!
```

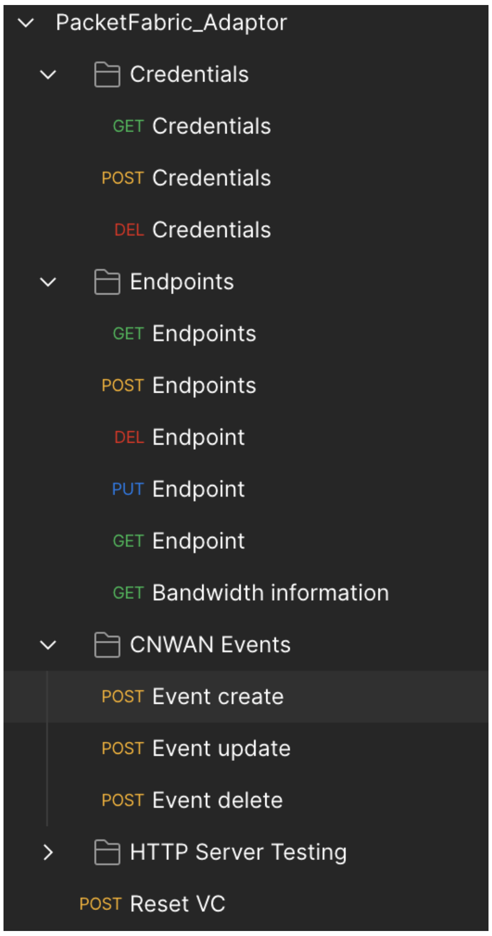
						INTERFACE A			INTERFACE Z		
TREND	TYPE	CIRCUIT ID	STATUS	SPEED	DESCRIPTION	LOCATION	STATUS	VLAN	LOCATION	STATUS	VLAN
	Backbone	PF-BC-SEA-WDC-1701620-PF	Active	200Mbps	Seattle to Washington - SD-WAN premium link	SEA/SEA1/WX-SEA1	Active	14	WDC/WDC2/CS-VA1	Active	14
	Backbone	PF-BC-WDC-CHI-1700195-PF	Active	1Gbps	Ashburn to Chicago - Video Traffic	WDC/WDC2/CS-VA1	Active	100	CHI/CHI1/EQ-CH1	Active	100
	Backbone	PF-BC-CHI-WDC-1700190-PF	Active	1Gbps	Chicago to Ashburn - User Traffic	CHI/CHI1/EQ-CH1	Active	1000	WDC/WDC2/CS-VA1	Active	1000

© 2021 PacketFabric, Inc. Legal

Figure 8: KubeConEU ScaleX demo presented by PacketFabric

C API endpoints

Figure 9 shows all the endpoints implemented in the CN-WAN Adaptor API.


PacketFabric_Adaptor

Credentials

GET Credentials

POST Credentials

DEL Credentials

Endpoints

GET Endpoints

POST Endpoints

DEL Endpoint

PUT Endpoint

GET Endpoint

GET Bandwidth information

CNWAN Events

POST Event create

POST Event update

POST Event delete

HTTP Server Testing

POST Reset VC

Figure 9: CN-WAN Adaptor API endpoints