TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 01/10 pp 635–644
DOI: 10.26599 / TST.2018.9010062
Volume 23, Number 6, December 2018

MN-SLA: A Modular Networking SLA Framework for Cloud Management System

Zhi Liu, Shijie Sun, Ju Xing, Zhe Fu, Xiaohe Hu, Jianwen Pi, Xiaofeng Yang, Yunsong Lu, and Jun Li*

Abstract: With the proliferation of cloud services and development of fine-grained virtualization techniques, the Cloud Management System (CMS) is required to manage multiple resources efficiently for the large-scale, high-density computing units. Specifically, providing guaranteed networking Service Level Agreement (SLA) has become a challenge. This paper proposes MN-SLA (Modular Networking SLA), a framework to provide networking SLA and to enable its seamless integration with existing CMSes. Targeting at a modular, general, robust, and efficient design, MN-SLA abstracts general interacting Application Programming Interfaces (APIs) between CMS and SLA subsystem, and it is able to accomplish the integration with minor modifications to CMS. The evaluations based on large scale simulation show that the proposed networking SLA scheduling is promising in terms of resource utilization, being able to accommodate at least 1.4× the number of instances of its competitors.

Key words: networking; service level agreement; cloud management system

1 Introduction

As the foundation of the multi-tenancy cloud, Cloud Management System (CMS) takes the responsibility of managing the resources of cloud infrastructure and provisioning the computing instances for tenants. Server virtualization technologies have enabled the fine-grained slicing and allocation of resources like CPU, memory, and disk[1], while networking resource allocation has always been a challenge in multi-tenancy cloud. On one hand, logically neighboring instances of the same tenant could be scattered across multiple servers, but should be interconnected as if they were in the same Local Area Network (LAN)[2]. Besides, physically neighboring instances of multiple tenants should be strictly isolated to guarantee security, even though they are located within the same server. Efforts in both academia and industry have been addressing the networking problem by leveraging Software Defined Networking (SDN) technique and its most popular application—network virtualization[3].

Taking an in-depth analysis from the perspective of resources, it could be observed that resources like CPU, memory, disk, and even cache[4] could possibly be provisioned with Service Level Agreement (SLA), claiming the reservation of certain resource quantity and quality. However, current CMSes have not been able to address the SLA for network resource, leaving multiple tenants to share the physical bandwidth in an unrestrained manner. One might imply that network may not be as busy as CPU or memory, and will be less noticeable during

- Zhi Liu, Shijie Sun, Ju Xing, Zhe Fu, and Xiaohe Hu are with Department of Automation, Research Institute of Information Technology, Tsinghua University, Beijing 100084, China. E-mail: zhi-liu12@mails.tsinghua.edu.cn; ssj13@mails.tsinghua.edu.cn; xingj15@mails.tsinghua.edu.cn; fu-z13@mails.tsinghua.edu.cn; hu-xh14@mails.tsinghua.edu.cn.
- Jun Li is with Research Institute of Information Technology, Tsinghua National Lab for Information Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: junl@tsinghua.edu.cn.
- Jianwen Pi, Xiaofeng Yang, and Yunsong Lu are with Huawei Inc., Santa Clara, CA 95050, USA. E-mail: Jianwen.Pi@huawei.com; Karl.Yang@huawei.com; Yunsong.Lu@huawei.com.
- * To whom correspondence should be addressed.
Manuscript received: 2017-01-08; accepted: 2017-02-22

the peak time. However, a recent shift from traditional Virtual Machine (VM) based service to container based "microservice" significantly aggravated the networking problem. Containers are lightweight and could be provisioned 1–2 orders more than the number of virtual machine on single machine[5], making the networking resource even more constrained.

This paper addresses the problem by proposing MN-SLA, a Modular Networking SLA framework for the CMS. MN-SLA abstracts general interacting Application Programming Interfaces (APIs) between CMS and SLA subsystem, and it is able to accomplish the integration with minor modifications to CMS. Also, the framework is able to provide networking SLA under different SLA enforcement modes. Preliminary evaluation results show that the proposed SLA scheduling algorithm is promising in terms of network resource utilization and is able to accommodate at least 1.4× the number of instances of its competitors. The rest of this paper is organized as follows: Section 2 gives the background of networking SLA in multi-tenancy cloud and summarizes the design challenges of SLA subsystem; Section 3 introduces the architecture and design of MN-SLA, and elaborates how this framework addresses the design challenges. The proposed SLA scheduling algorithms are illustrated in Section 4, followed by implementation and evaluation in Section 5. In Section 6, we conclude this paper as well as discuss about the future work.

2 Background

Currently, most services of cloud are based on VMs. As container technology moves into maturity and application deployment in containers has become automated[6, 7], container-based cloud services become promising and are already adopted by many top cloud providers[8, 9] in their services. There are also scenarios which require both VM and container deployment[10]. Therefore, in this paper, we discuss the SLA solution for general computing units, and use the term "instance" to represent both VM and container.

Networking SLA in this paper is taken mostly as network bandwidth SLA, this is due to the fact that other network measurements like latency is more non-deterministic, and is affected by many factors such as queuing in software and hardware switches[11], application response, and server IO, etc. From the perspective of users, the SLA intentions are specified based on a "service" model as shown in Fig. 1. Instances are categorized by services based on their functionality, and services could be interconnected with internet or other services by logical "linking". Networking SLA parameters could be specified on the above linkings. For example, a cloud tenant might specify certain amount of bandwidth to be reserved between his web service and database service. However, inter-service SLA is not straight-forward to implement since the number of instances in each service could change as the service scales horizontally. Therefore, this paper considers two schemes of SLA specifications, and either of them maps the inter-service SLA into inter-instance SLA differently.

The first one is "fixed inter-instance bandwidth". The bandwidth on the linking represents that such amount of bandwidth should be reserved between any pair of instances of interconnected services. The second scheme is "fixed aggregated inter-service bandwidth". In this case, the numbers on linkings represent the aggregated bandwidth between the services, and the inter-instance bandwidth is derived by dividing the aggregate bandwidth equally among all of the instance pairs. With such fine-grained inter-instance SLA, the dataplane policy entries could be generated and enforced to the appropriate switches.

The SLA framework should be able to manage network resources efficiently, and should also integrate with

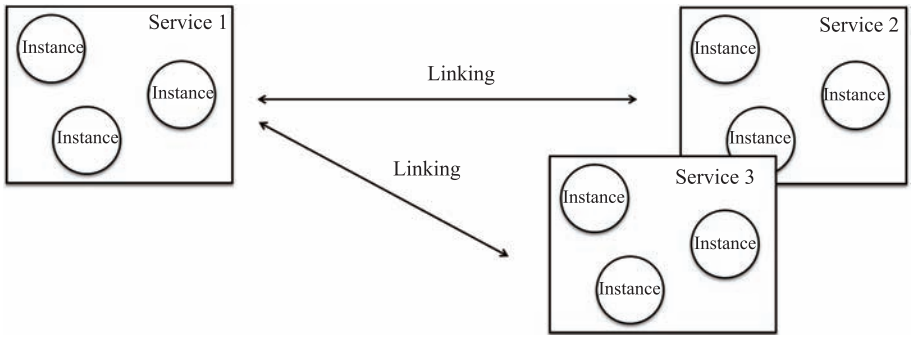


Fig. 1 SLA user definition model.

existing CMSes seamlessly. Currently, there have been varieties of CMS in production and it would be too disruptive to re-factor them in order to add the networking SLA functionality. In addition, CMS is the final decision point of instance deployment and maintains information essential to SLA enforcement. Therefore, there should be mechanisms for such information to be exchanged and synchronized between CMS and SLA subsystem. Given that different CMSes might consume the SLA functionalities at different stages during the instance provisioning, the design should be adaptive to these uncertainties.

To summarize, the design of the SLA subsystem should meet the following requirements.

• **Modularity**. The SLA subsystem should be a separate module and loosely coupled with CMS, and the integration should require minor modifications to the CMS.

• **Generality**. Different cloud management systems should call standard APIs to interact with SLA subsystem. These abstracted APIs should be able to deliver a wide range of SLA functionalities.

• **Robustness**. The SLA subsystem should be adaptive to different SLA enforcement modes, and should be able to handle the uncertainties of CMS to deliver the same SLA functionality.

• **Efficiency**. The SLA subsystem should carefully schedule the instance placement based on network resource, so that the CMS is able to accommodate more tenant instances.

3 MN-SLA Design

To meet the above design requirements, MN-SLA makes several design choices.

Separation of basic networking and SLA. Given that basic networking functionalities have already been incorporated by most existing CMSes, MN-SLA leaves basic networking unchanged in existing CMSes and only handles networking SLA functionalities. However, this also brings about the problem that the SLA subsystem is unaware of the locations, affiliations, and networking configurations of each instance, without which the SLA policies cannot be derived. Therefore, several APIs are designed for CMS and SLA subsystem to synchronize such information. One might think that this requires plenty of changes in CMS, but what is found indicates that many CMS implementations[7, 12] rely on some reliable datastore to keep their configurations for robustness. Such datastore is a perfect synchronization point, where SLA subsystem could be registered as a listener of any changes of instance configurations.

Offer-based scheduling. Generally, the CMS manages resources like CPU and memory to figure out the location for instance deployment. When integrating networking SLA subsystem with CMS, one important problem is how to incorporate network-oriented scheduling with the CMS placement. MN-SLA addresses this problem based on "offer-based scheduling". On each physical machine within the management domain, a CMS agent will report to the CMS its available resource in form of "offer". The CMS then sorts these offers according to the provisioning purposes, e.g., minimizing the overall maximum workload. To integrate with the SLA subsystem, the SLA plugin extracts these offers and consults the SLA subsystem for placement solutions considering network resource. The networking-oriented solution will be incorporated with results based on other resources (e.g., CPU and memory, etc.) to determine the final deployment location.

3.1 Architecture and components

The overall architecture of our MN-SLA design and components are shown in Fig. 2.

• **SLA plugin.** The SLA plugin integrates the networking SLA functionalities with the CMS. It parses specifications from tenants and extracts the networking SLA intentions. In addition, it intercepts the offers of the CMS and consults the SLA subsystem for evaluation based on networking resource.

• **API handler.** The API handler receives Representational State Transfer (REST) requests from CMS, and extracts the type and payload of each request. The handler then calls the related modules in SLA subsystem to execute the request, and finally replies with the result returned.

• **Deployment config manager.** The deployment config manager is designed to get instance configurations from CMS, so that the SLA subsystem could be provided with the information like locations, affiliations, and networking configurations.

• **SLA scheduler.** The SLA scheduler is responsible for implementing the SLA sorting and adjustment algorithms, providing the evaluating or relocating results based on current network conditions.

• **SLA policy manager.** SLA policy manager handles the policy enforcement requests. It generates dataplane SLA policy entries and delegates the policy renderer to

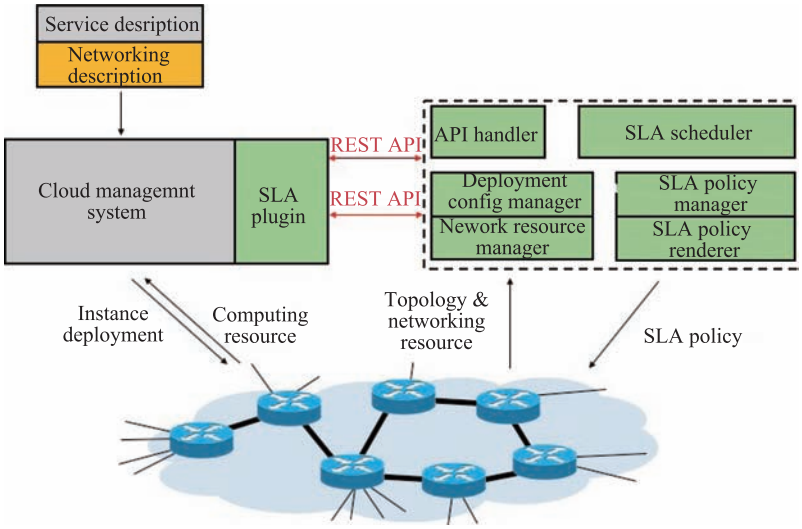


Fig. 2 Overall architecture of SLA subsystem.

push the policies to dataplane. Meanwhile, it maintains both the logical and dataplane policy entries in a database to simplify further policy modifications.

• **Network resource manager.** Network resource manager collects topology and link capacities from the underlying infrastructure. It also constructs a "network resource graph", which will be consumed by other modules for purposes such as calculating the capacity between instances or determining the policy enforcement point.

• **Policy renderer.** The policy renderer implements the actual dataplane policies to the OpenFlow-enabled switches. Since the implementing of SLA policy requires multiple southbound protocols, it hides the underlying complexity and provides a clean interface for SLA policy enforcement.

3.2 Execution modes and APIs

The SLA subsystem is designed to support user's SLA specification under different SLA enforcement modes. One important factor is the sequential order of SLA definition, i.e., user might define SLA policies either before or after the target instance is deployed, which has great impact of how networking SLA should be enforced. In the first case, the location of the target instance is carefully picked to avoid over-congestion. While in the latter case, the instance has been deployed before networking SLA is specified and the SLA enforcement might fail. Therefore, the SLA subsystem is designed to support two modes to accommodate SLA intentions—proactive mode and passive mode. To support the implementation of the above two modes, a group of standard APIs are designed, as listed in Table 1.

3.2.1 Proactive mode

In proactive mode, a user defines the services with the intermediate networking SLA specification. When the user submits a request to deploy an instance, the SLA plugin extracts the offers and SLA specifications from the CMS, and issues an SLASortingRequest() to the SLA subsystem. On receiving the request, the service affiliation and its SLA peers could be derived. The SLA subsystem returns with offers sorted according to their satisfaction of the SLA specifications. The CMS then picks the final location to deploy this instance, and passes the deployment information via InstanceDeployInfo() to the SLA subsystem. Finally, the CMS calls the SLAPolicyEnforcement() to delegate the SLA subsystem for enforcing the corresponding SLA policies.

3.2.2 Passive mode

In passive mode, the CMS first deploys the target instances without calling SLASortingRequest() beforehand. It then

Table 1 SLA APIs.

API	Description
SLASortingRequest (SLAIntention, offers)	Consult SLA subsystem for ordered offer according to networking SLA
InstanceDeployInfo (InstanceInfo)	CMS passes instance deployment information to SLA subsystem
SLAPolicyEnforcement (InstanceSLASpec)	CMS delegates the SLA subsystem to enforce the dataplane policies
SLAAdjustmentRequest (SLAIntention, offers)	CMS delegates the SLA subsystem to release network resource for current SLA intention

issues an SLAPolicyEnforcement() to push the SLA polices. Since the capacity of links can be inadequate due to non-optimized placement, the enforcement request will return with a failure. In this case, the CMS will issue an SLAAdjustmentRequest() to ask for instance relocation, in order to release enough bandwidth. The SLA subsystem then returns with adjustment (i.e., instance relocation) solutions and the CMS picks one. The CMS then migrates the instances and informs the SLA subsystem by calling InstanceDeployInfo(). The SLA subsystem will update the corresponding SLA dataplane policy entries based on the new instance locations. Finally, the CMS will issue another SLAPolicyEnforcement() after the adjustment.

4 SLA Scheduling Algorithms

The SLA scheduling algorithm is responsible for providing solutions of instance placement to CMS based on current network capacity. The scheduling of networking resource is different from other resources like CPU and memory, which only need to consider the local resource fulfillment. The evaluation of each offer requires checking the paths between all inter-connecting peers, and the capacity of each path relies on all its belonging links. Therefore, the scheduling algorithm requires complex calculation based on the topology, network resource, and instance locations.

There are a couple of works that focussed on network resource scheduling. Each of them addresses the problem from different perspectives. Faircloud[13] and NeTShare[14] proposed several different models to divide the network resource when bandwidth is insufficient. However, these solutions focus on the fairness among multiple flows or tenants when the network is over-congested, and do not provide guaranteed networking SLA for tenants.

Oktopus[15] proposed the abstraction of "virtual cluster" to describe the network resource requirements. It proposed a greedy algorithm that finds the lowest topology hierarchy to accommodate the whole virtual cluster and reserve network resource for its inter-connected instances. However, the placement is done in granularity of cluster, which indicates that the tenants should be aware of their whole network structure and requirements beforehand. Such assumptions do not hold since cloud tenants mostly add instances incrementally as their services scale up. On the other hand, Ref. [16] schedules the network resource in granularity of Jobs/Tasks, and considers the time varying characteristic of network utilization. It tries to interleave high-demand workloads with low-demand ones, and thus achieves high link utilization. Such proposal works for workloads like batch computing tasks, but not for tenants that occupy the instances and network bandwidth for a long time. In addition, it requires the networking utilization profile of various tasks, which is hard to acquire and may change quickly in production systems. Therefore, it could be observed that existing solutions do not support guaranteed networking SLA in an incremental manner, and new scheduling abstractions and algorithms are required to address the problem.

On the other hand, the scheduling algorithm should prevent complex calculation, in order to return the result within a predictable amount of time. To accelerate the computation, the algorithm is designed to rely on some pre-calculated structures, i.e., the Network Resource Matrix (nr_matrix), the Distance Matrix (d_matrix), and the Network Resource Graph (nr_graph). These structures store the information for each "Deployment Entity(DE)", which is the lowest level entity that could instantiate an instance. In the following discussions, the term "DE" refers to a host in our physical setup. (A DE can also be a VM in some scenarios, where containers are considered to be deployed on such DEs.)

• **Network resource matrix** stores the available network resource between any DEs, it is an $N \times N$ matrix where N is the number of DEs in the system. For each DE pair, the corresponding value is the available network resource (bandwidth capacity) along their interconnecting path. Though generating paths for all DE pairs is time-consuming, the paths could be generated only when the topology changes. The bandwidth capacity is incrementally updated after each bandwidth allocation, enabling fast calculation of the evaluating results.

• **Distance matrix** stores the distances between any DEs, and also has a size of $N \times N$. The distance is counted in terms of number of hops between the DEs. The longer the distance is, the more links it occupies. When the SLA sorting algorithm evaluates the offers according to locality, it consults this distance matrix for each inter-connected pair. Also, such distance information is essential for further enforcing networking latency SLA, which will be included in our future work.

• **Network resource graph** is constructed by the network resource manager. The graph is consulted to generate the path and the available capacity between any DE pair. The construction and update of distance matrix and network resource matrix also rely on this graph.

In addition, cloud data center networks are always constructed with hierarchy[17], where high-layer links carry a larger amount of aggregated traffic and are prune to

congestion[18]. Thus localization is an important concern to relieve the pressure of high-layer links, as well as to reduce the communication latency. Therefore, distance and overall link utilization are two important metrics for our heuristic. The goal of our SLA scheduling algorithm is to introduce least load increase to links, and to place the communicating instances as close as possible.

4.1 SLA sorting algorithm

Given the offers to be evaluated and user's SLA intention, the offers are sorted according to the network capacity. Since CMS might deliver a large number of offers to be evaluated at a time, which might result in unpredictable response time, the sorting algorithm is designed to return at most *C* offers. The algorithm firstly sorts the offers based on the distance between the interconnecting instances, then filters out the offers that cannot meet the SLA specifications, and finally evaluates each offer and returns at most *C* feasible offers according to its min-max optimization goal, then reserves the network resource based on the SLA specification.

As shown in the pseudo code, the evaluating algorithm (Algorithm 1) is conducted in the following steps.

Algorithm 1 SLA sorting algorithm

```
1: function SLASORTINGALGORITHM (sla_intention, offers, c)
2:     ▷ Sort the offers by total distance of peers
3:     offers = SortByDis (offers, d_matrix)
4:     ▷ Evaluate each offer until getting c feasible offers
5:     feasible_num = 0
6:     for ofr in offers do
7:         cost[ofr] = 0
8:         is_feasible = 1
9:         DeployInstance (r_graph, nr_matrix, sla_intention, ofr)
10:        for conn in sla_intention do
11:            ▷ Remove the offers exceeding bandwidth limit
12:            if nr_matrix[ofr][conn['peer']] > 1.0 then
13:                is_feasible = 0
14:                offers.remove (ofr)
15:                break
16:            end if
17:            ▷ Update the cost of the offer
18:            cost[ofr] += nr_matrix[ofr][conn['peer']]
19:        end for
20:        feasible_num += is_feasible
21:        RemoveInstance (r_graph, nr_matrix, sla_intention, ofr)
22:        ▷ Only keep the first c offers
23:        if feasible_num ⩾ c then
24:            break
25:        end if
26:    end for
27:    ▷ Sort the feasible_num offers by cost
28:    offers = SortByCost (offers, cost, feasible_num)
29:    return offers
30: end function
```

4.1.1 Offer sorting

For each candidate offer, the algorithm calculates the total distance between the interconnected pairs involved in this SLA intention. The calculation directly refers to the values in the distance matrix and the complexity will be $O\ (N \times P)$, where N is the number of offers and P is the number of peering instances. Later, the algorithm sorts the offers according to this distance, resulting in the offer with least total distance to appear first after sorting. The intuition is that the offers with better locality will be further evaluated first, and the complexity of the sorting will be $O\ (N \times \log N)$.

4.1.2 Offer filtering

Since SLA policies are specified in a high-level manner (e.g., service-to-service), each offer requires evaluating the network resources between multiple peering DEs. The algorithm consults the deployment config manager to determine which instances are related to the policy and convert the original inter-instance demands to corresponding inter-DE demands. The network resource matrix will be consulted to filter out the offers without enough network resource. Since each peering DE could be filtered with $O(1)$ complexity, the fulfillment of each offer could be derived within $O(P)$. If none of the DE meets the SLA intention, the algorithm will return with an empty list indicating the network capacity is insufficient.

4.1.3 Offer evaluating

To save the time for evaluating, the algorithm evaluates at most the first C feasible offers. For each offer, the algorithm places the instance and adds the load to the involved paths. Then the algorithm calculates the cost of this offer according to the remaining capacity of the related links. The cost is generated by summing up the max link utilization ratio of each involved path. Finally the returned offers are sorted by cost in the ascending order.

4.2 SLA adjustment algorithm

The SLA adjustment algorithm (Algorithm 2) is called when processing an SLAAdjustmentRequest, following the failure of previous SLASortingRequest or SLAPolicyEnforcement. Given the desired SLA intention, the goal of SLAAdjustmentAlgorithm is to generate several instance relocating solutions so that the SLA intention could be accommodated. The SLA intention is included in the SLAAdjustmentRequest, which should be the same as previous failed SLASortingRequest. The SLA adjustment algorithm runs in the following steps.

Algorithm 2 SLA adjustment algorithm

```
1:  function SLAADJUSTMENTALGORITHM
    (sla_intention, offers, c)
2:      for ofr in offers do
3:          exceeded_traffic[ofr] = 0
4:          DeployInstance (r_graph, nr_matrix, sla_intention, ofr)
5:          for conn in sla_intention do
6:              ▷ Compute total exceeded traffic
7:              if nr_matrix[ofr][conn['peer']] > 1.0 then
8:                  path = r_graph.FindPath (ofr, conn['peer'])
9:                  exceeded_traffic[ofr]+=path.ExceededTraffic()
10:             end if
11:         end for
12:         RemoveInstance (r_graph, nr_matrix, sla_intention, ofr)
13:     end for
14:
15:     ▷ Sort offers by exceeded traffic and keep the best c offers
16:     offers = SortByExceededTraffic (offers, exceeded_traffic)
17:     offers = offers[0 : c]
18:     relocate_solutions = RelocationFind (sla_intention, offers)
19:     return offers, relocate_solutions
20: end function
21:
22: function RELOCATIONFIND (sla_intention, offers, c)
23:     for ofr in offers do
24:         relocate_solutions[ofr] = [ ]
25:         DeployInstance (r_graph, nr_matrix, sla_intention, ofr)
26:         no_solution = False
27:         for conn in sla_intention do
28:             if nr_matrix[ofr][conn['peer']] > 1.0 then
29:                 e_links = FindExceededLinks (r_graph,
                        ofr, conn['peer'])
30:                 for lk in e_links do
31:                     cncts = SortByContribution
                            (r_graph, GetConnections(lk))
32:                     solution = TryToRelocate (r_graph, cncts)
33:                     if solution == [ ] then
34:                         no_solution = True
35:                         offers.remove(ofr)
36:                         break
37:                     end if
38:                     relocate_solutions[ofr].append(solution)
39:                 end for
40:                 if no_solution then
41:                     break
42:                 end if
43:             end if
44:         end for
45:         Reverse (r_graph, relocate_solutions[ofr])
46:         RemoveInstance (r_graph, nr_matrix, sla_intention, ofr)
47:     end for
48: end function
```

4.2.1 Offer sorting

For each offer, the algorithm assumes that the target instance could be accommodated on that location and calculates the bandwidth capability of all links after deploying the instance with networking SLA. With these link utilizations related to this offer, the algorithm calculates the total amount of exceeding link capacity, indicating the amount of network resource to be released for this offer. After completing the calculation of each offer, the algorithm sorts the offers in the ascending order. Therefore, the offer that requires the least resource to release will be considered first in following steps.

4.2.2 Instance relocating

The algorithm then iterates the sorted offers, trying to generate a relocating solution for each of them. Each relocating solution may migrate multiple instances. To generate the relocation solution for certain offer, the algorithm first checks the capacity of all links after enforcing the SLA for this offer. The over-congested links will be derived and the algorithm then tries to relocate some instances occupying the link to release enough bandwidth. To be more specific, the algorithm iterates the over-utilized links, and derives all the instance pairs that traverse those links. For each over-congested link, the corresponding instance pairs are sorted in descending order according to their contribution to the link congestion. Then the algorithm will try to move the instance pairs in order to mitigate existing over-congested links and not to introduce additional over-congestions.

If the replacement successfully releases enough capacity for all the over-utilized links, then a relocation solution is generated for the offer. Otherwise the algorithm continues with the next offer. To limit the calculation complexity, the number of offers that the algorithm iterates could be configured, which constrains the response time.

5 Implementation and Evaluation

MN-SLA prototype is implemented on OpenDaylight[19] controller, behaving as a controller application plugin. Link Layer Discovery Protocol (LLDP)[20] is used for topology discovery, where the SDN controller instructs an openflow-enabled switch to send an LLDP packet from a certain interface, and discovers the outgoing interface of the same packet to identify a link between switches. A consistent datastore is used to synchronize the instance deployment information. The deployment config manager is registered as a listener of this datastore, and will trigger its callback function on any data updates. Current prototype reserves link bandwidth by using the queuing capacities of switches. The policy renderer calls the OVSDB protocol to setup queues in the openflow-enabled switches and configures the desired bandwidth parameters.

It also invokes openflow protocol to direct the target traffic into these queues. In the prototype, the MN-SLA is integrated with a container management system—Kubernetes[7] and the SLA plugin is implemented as an extension of Kubernetes scheduler.

The SLA scheduling algorithm is evaluated by large-scale simulations. A simulator is developed to evaluate different algorithms, modeling a cloud datacenter with a three-tier network topology (i.e., ToR, aggregation, and core)[21]. All the simulations are run with 1 000 hosts, and each host can accommodate 4 instances. In addition, we test the algorithms with different networks over subscription ratio of 1:16 and 1:20, respectively. The input placement requests are generated according to the model similar to Fig. 1. Each user may request a network interconnecting 2–4 concatenated services, and each service may contain 5–15 instances. The input is generated at the scale of 300 users and is randomly re-ordered to imitate the evolvement of user services. The input requests are organized so that each request only contains one instance, and the length of the input traces is the maximum number of instances (4 000) of the simulated system.

Our scheduling algorithms have two variations, the first one (scheduling) processes the input placement requests with SLA sorting algorithm. The algorithm adopts the best offer if there is enough bandwidth capacity, otherwise the request fails and the algorithm turns to the next request. The second algorithm (with adjustment) inherits the first algorithm, and extends it with further scheduling. If the network resource is not adequate to accommodate the requests, the SLA adjustment algorithm is called to release network resource and the first adjustment solution is adopted to deploy the instance. If the adjustment succeeds, the algorithm will then retry the placement.

These proposed algorithms are compared against random placement and nearest placement algorithm. For random placement, it first selects all the available offers with enough capacity and then randomly picks a location among the available ones. Likewise, the nearest placement algorithm chooses the location with the least total distance among the available offers. Figures 3 and 4 show the results under different network oversubscription ratios. The y-axis represents the ratio of accepted requests, and each column depicts the distribution with 100 input traces. It is observed that the median number of accommodated instances for our first scheduling algorithm is 1.4×–2.3× that of the nearest placement algorithm, and is 15× that of the random placement algorithm. Especially, the proposed

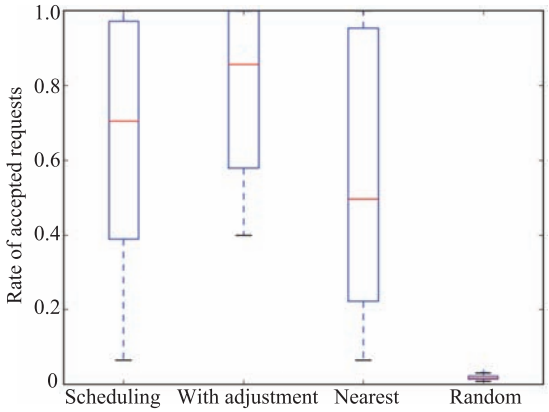


Fig. 3 Scheduling results with 1:16 network oversubscription.

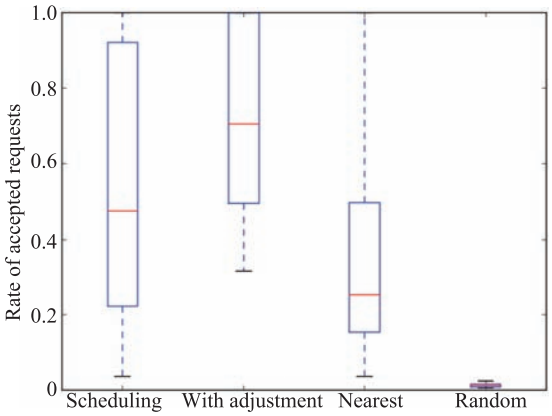


Fig. 4 Scheduling results with 1:20 network oversubscription.

algorithm achieves more improvement in networks with higher oversubscription ratio. In addition, the adjustment heuristic achieves the improvement of 21%–51%, indicating that the adjustment algorithm is able to support incremental networking SLA enforcement and is promising to further improve the resource utilization.

6 Conclusion and Future Work

This paper introduces MN-SLA, a modular networking SLA framework for the CMS. MN-SLA is able to be integrated with existing CMSes with minimum efforts. The evaluation results show that the proposed networking SLA scheduling algorithm is able to accommodate at least 1.4× the number of instances of existing solutions and is promising in terms of network resource utilization. Our future work includes deployment of MN-SLA in production environment for the evaluation of its practical scalability and efficiency. Also, adding network latency into the SLA scheduling matrix and incorporating dynamic network status are also considered.

References

[1] G. Boss, P. Malladi, D. Quan, L. Legregni, and H. Hall, Cloud computing, *IBM White Paper*, vol. 321, pp. 224–231, 2007.

[2] X. Wang, Z. Liu, Y. X. Qi, and J. Li, Livecloud: A lucid orchestrator for cloud datacenters, in *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, 2012, pp. 341–348.

[3] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, et al., Network virtualization in multi-tenant datacenters, in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 203–216.

[4] V. Shivappa, An introduction to cache quality of service in linux, http://events.linuxfoundation.org/sites/events/files/slides/presentlinuxcon_vikas_0.pdf, 2017.

[5] R. Rosen, Linux containers and the future cloud, *Linux J*, no. 240, 2014.

[6] Swarm, http://www.docker.com/products/docker-swarm, 2017.

[7] Kubernetes, http://kubernetes.io/, 2017.

[8] Amazon, https://aws.amazon.com/ecs/, 2017.

[9] Azure, https://azure.microsoft.com/en-us/services/container-service/, 2017.

[10] Virtual machines and containers in azure, https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-linux-containers/, 2017.

[11] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda, Less is more: Trading a little bandwidth for ultra-low latency in the data center, in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012, pp. 253–266.

[12] Openstack, https://www.openstack.org/, 2017.

[13] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, Faircloud: Sharing the network in cloud computing, in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2012, pp. 187–198.

[14] T. Lam, S. Radhakrishnan, A. Vahdat, and G. Varghese, NetShare: Virtualizing data center networks across services, University of California, San Diego, Technical Report CS2010-0957, May. 2010.

[15] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, Towards predictable datacenter networks, in *ACM SIGCOMM Computer Communication Review*, 2011, pp. 242–253.

[16] D. Xie, N. Ding, Y. C. Hu, and R. Kompella, The only constant is change: Incorporating time-varying network reservations in data centers, *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 199–210, 2012.

[17] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, Vl2: A scalable and flexible data center network, in *ACM SIGCOMM Computer Communication Review*, 2009, pp. 51–62.

[18] J. Mudigonda, P. Yalagandula, J. Mogul, B. Stiekes, and Y. Pouffary, Netlord: A scalable multi-tenant network architecture for virtualized datacenters, in *ACM SIGCOMM Computer Communication Review*, 2011, pp. 62–73.

[19] Opendaylight, https://www.opendaylight.org, 2017.

[20] P. Congdon, Link layer discovery protocol, RFC 2922, July. 2002.

[21] T. Benson, A. Akella, and D. A. Maltz, Network traffic characteristics of data centers in the wild, in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, 2010, pp. 267–280.

Yunsong Lu is chief architect of networking at Huawei North America R&D Center, and he is currently leading research projects on AI-driving networking, cloud networking, and programmable micro data plane etc. At Huawei, he founded virtual networking lab where he created Elastic Virtual Switch (EVS), S-DNA (Software-Defined Network Acceleration), Canal (Container Networking Framework), and reactive network SLA technologies etc., which have been the fundamental building blocks of Huaweis Cloud, SDN, and NFV solutions. Previously, he worked on Solaris Networking at Sun Microsystems and Oracle, where he designed network resource management and offloading, frameworks and refactored solaris TCP/IP data path, etc.

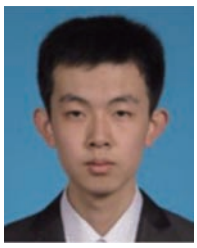
Jun Li received the BEng and MEng degrees from Tsinghua University, China in 1985 and 1988, respectively, and the PhD degree from New Jersey Institute of Technology in 1997. He is currently a professor at the Research Institute of Information Technology, Tsinghua University, China. He was a board director of XML global and is currently a board member or advisor of several venture capital and startup companies, including GigaDevice and Agate Logic. He is also a deputy director of the Tsinghua National Lab for Information Science and Technology. He is a co-author of more than 100 papers, and co-inventor of 10 patents. He is also a managing director of an angle fund versatile venture capital. His research interests include network security, pattern recognition, and image processing.

Zhi Liu received the BEng and PhD degrees from Tsinghua University, China in 2012 and 2017, respectively. He is currently a software engineer at Xiaomi Inc. His research interests include software-defined networking, high-performance packet processing algorithms, data center networking, and security.

Shijie Sun received the BEng degree from Tsinghua University, China in 2017. He is currently a software engineer in Didi Chuxing. His research interests include computer networking and machine learning.

Ju Xing received the BEng degree from Xidian University, China in 2015. He is currently a PhD student in Department of Automation at Tsinghua University, China. His research interests include software-defined networking, distributed systems, and network trouble shooting.

Zhe Fu received the BEng degree from Tsinghua University, China in 2013. He is currently a PhD student in Department of Automation at Tsinghua University, China. His research interests include cloud datacenter networks, deep inspection algorithms, and traffic shaping algorithms.



Xiaohe Hu received the BEng degree from Tsinghua University, China in 2014. He is now a PhD student in Department of Automation at Tsinghua University, China. His research interests include software-defined networking, cloud datacenter networks, network monitoring, and management.

Jianwen Pi is currently a software engineer and architect in Alibaba Groups. Prior to that he has architected and led developing many networking security and virtual networking products in different companies: Palo Alto networks, Huawei, and Juniper Networks, etc. He got the bachelor and MS engineering degrees from Huazhong University, China and University of British Columbia, respectively. His major research interests are distributed networking security and networking performance in data center.

Xiaofeng Yang is currently the director of research center of Yunshan Networks, Silicon Valley, US. He once worked at the Software Lab, Huawei R&D, USA. His research interests include network virtualization, parallel and distributed computing for SDN controller, and traffic shaping. He received the BEng and master degree from Tsinghua University, China. He has been working on big data analytics for cloud networking behavior in the past several years.