Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Quokka: Latency-Aware Middlebox Scheduling with dynamic resource allocation

Qing Li[a], Yong Jiang[a,*], Pengfei Duan[a], Mingwei Xu[b], Xi Xiao[a]

[a] Graduate School at Shenzhen, Tsinghua University, Shenzhen, Guangdong, China
[b] Tsinghua University, Beijing, China

ARTICLE INFO

Keywords:
Software defined networks
Network function virtualization
Middlebox

ABSTRACT

In the current Internet, middlebox management has become a significant challenge for network operators. Some schemes based on Software-Defined Networking (SDN) have been proposed in academia to simplify middlebox scheduling. However, these schemes are inefficient with the dynamical traffic requirements, as they mainly focus on stationary hardware middleboxes. Furthermore, the latency cannot be guaranteed in these schemes. In this paper, motivated by Network Function Virtualization (NFV), we propose the scheme of Quokka to solve these problems. In Quokka, we build a management framework for both computing resource and flow latency. We also design a latency model to describe the latency behaviour of flows. Based on the framework and model, we present a latency-aware scheme Quokka with portable software-based middleboxes that can be dynamically scheduled (*placed*) according to the changing traffic and resources. Quokka controls the latency by dynamically positioning the software middleboxes and scheduling the changing traffic. Therefore, different IT services with different latency requirements can be satisfied in Quokka. Comprehensive experiments show: (1) compared with traditional configuration methods, Quokka reduces the latency by about 20% on average; (2) Quokka requires 30–50% less middleboxes than traditional schemes to achieve the same performance.

1. Introduction

Middleboxes, for instance, firewalls, proxies and WAN optimizers, are networked appliances for complicated processing of packets. In the traditional TCP/IP network, the management of middleboxes is generally a significant challenge because of intricacy in configuring the flow service chain (Sherry et al., 2012; Quinn et al., 2014), which describes the ordered processing list of middleboxes for a specific flow. The configuration job becomes a nightmare especially when the policies are correlative (Qazi et al., 2013). Software-Defined Networking (SDN) is a new Internet paradigm that has and will continue to make significant influence on the network infrastructure (Casado et al., 2007; Nunes et al., 2014). As an innovative network technology, SDN has been proposed and well accepted for network resource management (Kim and Feamster, 2013). Specially, SDN can be used by the network operators to simplify middlebox configuration. In previous works (Qazi et al., 2013; Fayazbakhsh et al., 2014), SDN is employed for flexible middlebox management, which helps the operators reduce the complicated work of manual configuration.

However, these previous schemes only focus on generating middlebox-related policies or flow entries for stationary hardware middleboxes, which are inflexible in serving the dynamically changing traffic. In FlowTags (Fayazbakhsh et al., 2014 and Simple-fying (Qazi et al., 2013), middleboxes (the number and positions) are fixed after network initialization. Therefore, to make the packet passing a specific middlebox, a path stretch may occur, which may cause extra latency. Besides, the stationary deployment of hardware middleboxes cannot follow the pace of dynamic traffic changing. Contributions of these schemes mostly lie in middlebox configuration, and they pay less attention to flow scheduling, thus is poor at restricting the latencies of packets.

To overcome the inefficiency problem in resource utilization and latency control, we adopt Network Function Virtualization (NFV) technology (Chowdhury and Boutaba, 2010), and employ software middleboxes. In previous studies of NFV (Kohler et al., 2000; Dobrescu et al., 2009), particular attentions are devoted to network forwarding and processing in software. Based on these technologies, we build a framework to manage both computing resource and flow latency .

In this paper, based on prior NFV/SDN researches, we present Quokka,[1] a latency-aware and dynamic scheduling scheme of software middleboxes. Quokka efficiently places middleboxes at proper positions according to changing flows and assigns traffic to corresponding middleboxes in the required service chain. Quokka is both efficient

* Corresponding author.
E-mail address: jiangy@sz.tsinghua.edu.cn (Y. Jiang).
[1] Quokka is a kind of tiny and agile kangaroo in Western Australia, indicating flexibilities of software middleboxes.

http://dx.doi.org/10.1016/j.jnca.2016.10.021
Received 25 December 2015; Received in revised form 29 July 2016; Accepted 31 October 2016
Available online 05 November 2016
1084-8045/ © 2016 Elsevier Ltd. All rights reserved.

and reliable, as Quokka's scheduling algorithm reacts to dynamic flows robustly, thus continuously avoiding resource utilization inefficiency and providing the lower latency for end users.

More specifically, we make the following contributions:

(1) We propose a model to formulate the latency behaviours of packets, when they are being processed in the virtual middleboxes. Different types of middleboxes may operate in different ways. For instance, some middleboxes process the packet headers only, while others may have to handle the whole packets together with data payloads. Therefore, we employ two different queueing models to analyze their internal behaviours.
(2) We build a resource management framework, which employs both the resource master and agents to handle computing resource dynamically. Resource agents monitor resource usage status and report changes to the master. The master provides up-to-date resource snapshot to SDN controller for future usage in middlebox controlling and scheduling. Fault-handling and blacklist mechanisms are also developed to make the framework more stable.
(3) We design one algorithm to find the optimal positions for middleboxes, and another algorithm to schedule traffic with the given middlebox placement. Combining them together, Quokka can dynamically adjust the number and positions of middleboxes, and schedule flows in a low-latency way. Through these algorithms, we surmount the resource utilization inefficiency and the high delay problem in traditional scheduling schemes.

We evaluate the performance of our algorithms by comprehensive experiments with diverse topologies. Experimental results show: first, our scheduling algorithm averagely reduces the flow latency by 20% compared with the static middlebox configuration and the traditional load balancing; in addition, to achieve the same performance with the same resource restrictions, Quokka requires respectively 50% and 30% fewer middleboxes than the other two schemes.

2. Background and challenges

2.1. Middlebox management with SDN

Some schemes (Fayazbakhsh et al., 2014; Qazi et al., 2013) have been proposed to manage middleboxes using SDN technologies. However, contributions of these schemes mostly lie in middlebox configurations, and they pay less attention to flow scheduling, thus are poor at controlling the latencies of packets. The middlebox deployment is fixed after network initialization. Therefore, to make the packet passing a specific middlebox, a path stretch may occur, which may cause extra latency. Besides, the stationary deployment of hardware middleboxes cannot follow the pace of dynamic traffic changing.

Fig. 1 shows an example of the delay problem in traditional load balancing. Flows from $Host_0$ to $Host_1$ should be processed by a firewall, either $Firewall_a$ or $Firewall_b$. $Firewall_b$ is overloaded and the processing delay is 10 ms, while $Firewall_a$ is idle and the processing delay is only 1 ms. Under the traditional load balancing scheme, without considering the total packet delay, the flows will follow the path $Host_0 \rightarrow Firewall_a \rightarrow Host_1$, and the total latency should be 61 ms. However, taking the total latency into account, the packets walk through the path $Host_0 \rightarrow Firewall_b \rightarrow Host_1$, and the latency is only 35 ms. Consequently, these prior schemes are unreliable in latency restriction.

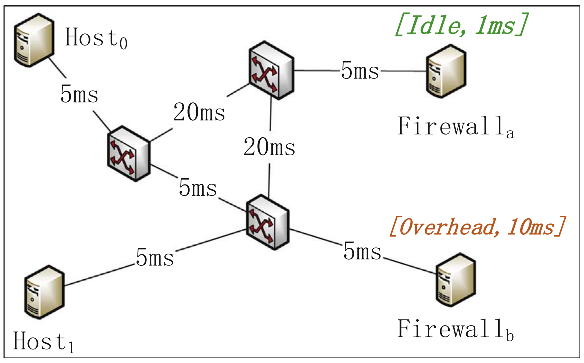

Fig. 1. Example: Load Balancing of Middleboxes.

2.2. NFV and benefits

Stratos (Gember et al.,) is proposed as a controllable and scalable framework for the efficient deployment of virtual middleboxes. OpenNF (Gember-Jacobson et al., 2014) enforces the functions of NFV with SDN, and provides a rich set of NFV/SDN APIs (*move*, *copy*, *share*, etc.) for software middleboxes management, which makes it feasible to dynamically schedule the middleboxes according to the changing traffic. ClickOS (Martins et al., 2014) is then put forward to improve the running efficiency of virtual middleboxes by optimizing the underlying Virtual Machines. ClickOS can launch the middlebox software within about 30 ms, which makes it possible for dynamically deployment (*add*, *delete*, etc.) of middleboxes according to online changing traffic.

2.3. Resource and latency management in clusters

Diverse applications are deployed in data centers and cloud platforms (Alizadeh et al., 2010). Generally the public clouds have the demand to provide the differentiated services for different applications from different customers (Amazon,). These applications include not only long time running batch (latency-insensitive) jobs like database backup/recovery and logging immigration, but also instant processing tasks, often tied with the user-facing products like mail and web search (Verma et al., 2015). The first kind of jobs, which we call *latency-insensitive jobs* here, often contain mass of data, but do not need realtime delivery; the second type, *latency-sensitive jobs*, usually carry less payload, but should be processed immediately. These flows should be handled separately with different priorities.

Latency management is critically important in these networks. It is reported that Amazon has 1% loss in revenue with 100 ms latency increasing and Google's data delivery rate drops down 20% when additional 500 ms latency is introduced (Munir et al., 2013). Traditional TCP/IP which provides best-effort service for flows, cannot provide any delay guarantees for high level applications. Previous works like DCTCP (Alizadeh et al., 2010) and D^2TCP (Vamanan et al., 2012) control sliding windows and make high priority scheduling for short-deadline flows in switches and routers.

As middleboxes are being introduced to various networks, including data centers (Joseph et al., 2008). In data center networks, load balancers and SSL offloaders are deployed for fast packet processing. These new coming appliances provide not only efficiencies but also new challenges for management. Both traditional methods and new architectures are proposed by prior researchers (Joseph et al., 2008; Qazi et al., 2013). Combing the middlebox management and latency management is of great challenge and critical value. Motivated by NFV technologies, we try to solve these two problems with a single management framework in Quokka.

3. Middlebox processing delay model

Our network environment is based on SDN, where there are four types of nodes: the controller, the switches, the hosts, and the resource pools. The resource pool is generally a multi-core server running a limited number of diverse middleboxes. To avoid the overheads of non-

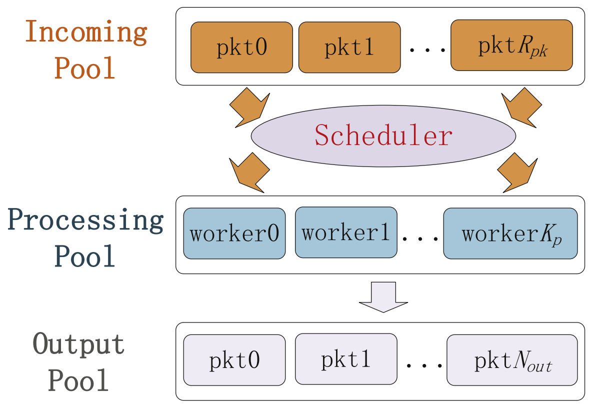

Fig. 2. The Middlebox Model with Multiple Workers.

uniform memory access (NUMA), we assume that a core is dedicated to a single middlebox (Mohammadkhan et al., 2015). Each link between nodes has a fixed transmission delay. The total delay of a packet includes the processing delay of middleboxes and the transmission delay of links in its path. In design of Quokka, to simplify the problem, we assume the link forwarding speed is much higher than the total processing speed of the pool. Accordingly, we fix the link transmission delay without considering the change caused by link queuing, as the inaccuracy is acceptable compared with the processing delay of middleboxes.

3.1. Model overview

We model the middlebox as a multiple-processor system with K_p processing units, called as workers. The packets arrive at the rate of R_{pk}. The middlebox schedules the K_p workers to process the arrived packets in the incoming pool. The middlebox model is shown in Fig. 2.

This model is motivated by some engineering backgrounds and previous works (Anderson et al., 2012; Ghodsi et al., 2012). xOMB (Anderson et al., 2012) is a general software middlebox framework. In xOMB packets or flows are buffered in queue and then scheduled to be handled by processing servers or units, which is consistent with the model we described above. When scheduling the processing units to deal with queued packets, we adopt a similar resource management technology as in Ghodsi et al. (2012) to schedule packets impartially in the processing units. In our model, we simply process packets in a First-Come-First-Service (FCFS) discipline. And each packet possesses a single processing unit during the manipulation.

To analyze packet delay in software middleboxes, an $M/M/c$ model is employed in Abdou et al. (2015), and we extend it in this paper. Furthermore, the serving behaviour in our queueing model is more complex, as we consider diverse software middleboxes. Therefore, we adopt two queueing models according to different processing behaviours of packets: *Deterministic Model* and *Exponential Model*.

In the *Deterministic Model*, the processing time of a packet is constant, not exponentially distributed as in Abdou et al. (2015). This is based on the fact that some middleboxes, for instance, proxies, firewalls, only process packet headers with fixed size. However, other middleboxes, like intrusion detection and prevention systems (IDS), process the whole packet, including the payload. If the data length is distributed exponentially, the serving time follows the Poisson Distribution. We use *Exponential Model* to analyze these kinds of middleboxes. In Abdou et al. (2015), the authors only employ *Exponential Model*, while both models are employed in this paper to formulate various middleboxes.

Subsequently, we will explain the two models in details. Before that, we give some definitions: (1) *waiting time*, denoted with W, is the time a packet queues in the incoming pool; (2) *processing time*, denoted with T, indicates the time required to process a packet by the processing unit without queueing. In the *Deterministic Model*, the *processing time* is constant, while in the *Exponential Model* it follows a Poisson Distribution; (3) *response time* S, also called *sojourn time*, is the sum of *waiting time* and *processing time*.

3.2. Deterministic model

In the *Deterministic Model*, T is the fixed serving time without queueing. Let $C_{mb} = \frac{K_p}{T}$ be the processing capacity of the middlebox. According to our model, the real processing rate ($ProcRate(R_{pk})$) of the middlebox can be computed by Eq. (1). Because one worker can handle a single packet at the same time, when $R_{pk} < C_{mb}$, the processing rate is R_{pk}, and the delay is T; when $R_{pk} \geq C_{mb}$, the incoming packets queue and are processed by the workers in the FCFS order. For the latter case, we compute the total delay of the packet according to the queueing theory.

$$ProcRate(R_{pk}) = \begin{cases} R_{pk}, & R_{pk} < C_{mb} \\ C_{mb}, & \text{else} \end{cases} \tag{1}$$

In this model, each host generates packets by Poisson Distribution $P(\lambda_1)$, and the arrival rate of all packets from n independent hosts follows the distribution of $P(\lambda_2)$, where $\lambda_2 = n\lambda_1$. Thus, the queueing model of this problem is $M/D/c$, a classical model in queueing theory ($c = K_p$ and $D{=}T$). The distribution of waiting time is provided as Eq. (2) (Gross, 2008).

$$F_W(x) = P(W \leq x) = \int_0^{\infty} F(t + x - T)\frac{\lambda_2^{K_p} t^{K_p-1}}{(K_p - 1)!}e^{-\lambda_2 t}dt \tag{2}$$

where F_W and P are the distribution and density function of waiting time W respectively. It can be further expressed as a piecewise function alternatively. As the following Eq. (3) shows, when $x \in [mD, (m + 1)D]$,

$$F_W(x) = P(W \leq x) = \sum_{n=0}^{K_p-1} \frac{\lambda_2^n e^{-\lambda_2}}{n!} \sum_{k=1}^{m} \frac{(-\lambda_2(x - kD))^{(k+1)K_p-1-n}}{((k + 1)K_p - 1 - n)!} e^{\lambda_2(x-kD)} \tag{3}$$

In *Deterministic Model*, the processing time of a packet, T, is constant. Suppose F_S is the distribution function of response time S. We have

$$F_S(x) = P(S \leq x) = P(W \leq x - T) \tag{4}$$

Suppose that the upper limit of response time in a middlebox is γ. Quokka guarantees the quality of transmission service by controlling the response time by a upper bound probability η according to Eq. (5).

$$P(S > \eta) = 1 - F_S(\gamma) < \eta \tag{5}$$

If flow behaviours (described by λ_1) and middlebox behaviours (described by K_p and T) are predefined, the processing delay of a middlebox is determined by the flow number n only, and we can solve Eq. (5) to get a maximal flow number. As for a chain of middleboxes, the distribution in Eq. (5) becomes the joint distribution of all the middleboxes in the service chain. Since we regard all the middleboxes are independent in states, the distribution can be expressed as Eq. (6).

$$F_S^{joint}(x) = \prod_{\sum_{i=1}^{k} x_i = x \wedge 0 \leq x_i \leq x} F_S(x_i) \tag{6}$$

where k is the length of the service chain, i.e., the number of middleboxes in this chain.

3.3. Exponential model

In *Exponential Model*, both the arriving time and serving time follow Poisson Distribution. In this subsection, we use similar symbols to those in *Deterministic Model*.

Suppose that T follows a Poisson Distribution $P(\mu)$, where

$E(T) = \mu$. Still, each host generates a $P(\lambda_1)$ flow, and n flows combines into a $P(\lambda_2)$ Poisson Distribution, where $\lambda_2 = n\lambda_1$. According to our model, the packets are served in the FCFS manner, thus the queueing model is $M/M/c$, which has also been deeply researched. We will give some properties of this model without detailed derivations, as they can be directly found in many queueing theory textbooks (Gross, 2008; Allen, 1978; Adan and Resing, 2002). Assume that the occupation rate per processing unit is $\rho = \frac{\lambda_2}{c \cdot \mu}$. The distribution of waiting time W is as Eq. (7) (Gross, 2008).

$$F_W(x) = P(W \le x) = 1 - P(W > x)$$
$$= 1 - \frac{(c\rho)^c}{c!}((1-\rho)\sum_{n=0}^{c-1}\frac{(c\rho)^n}{n!} + \frac{(c\rho)^c}{c!})^{-1}e^{-c\mu(1-\rho)x} \tag{7}$$

Similar to *Deterministic Model*, we can get some properties for response time S. However, T is not a fixed value in *Exponential Model*, but follows Poisson Distribution $P(\mu)$. Referring to results from Adan and Resing (2002), we have

$$F_S(x) = P(S \le x) = 1 - e^{-\mu x} - \frac{(c\rho)^c}{c!}((1-\rho)\sum_{n=0}^{c-1}\frac{(c\rho)^n}{n!} + \frac{(c\rho)^c}{c!})^{-1}$$
$$\frac{e^{-c\mu(1-\rho)x} - e^{-\mu x}}{1 - c(1-\rho)} \tag{8}$$

By the same technologies with the *Deterministic Model* part, we can get similar equations like Eqs. (5), (6).

4. Problem formulation

In Quokka, we choose to minimize the number of required NFs (middleboxes) with the constraints of flow requirements. We believe this is an energy-efficient approach. Besides, in a cloud providing virtual network service for enterprises, generally the middleboxes of each virtual network (VN) must be isolated from other VNs. In this scenario, minimizing the number of middleboxes in each VN can save the resources to support more VNs running on the physical infrastructure.

4.1. Problem with the uniform latency requirement

We first discuss the scenario where all the flows have the same upper limit latency. The problem is defined as following: given the topology, the traffic distribution and the maximal delay restriction, deploy the minimized number of middleboxes at the proper positions and schedule the traffic accordingly.

Let *src* and *des* be the source and destination of a flow. A *flow* is described as a triple $(src, dst, (MB_a, MB_b, MB_c))$, which means the flow from src to *dst* should be processed by middleboxes of type *a*, *b* and *c* in order. Let *FList* be the traffic flow list of a network. Let *PoolNum* be the number of resource pools for middleboxes in the network. Each pool can run *N* middleboxes at most. Let vector *pl* be a middlebox placement solution, where $pl[i]$ denotes the *i*th pool. Let $MBSet(pl[i])$ be set of middleboxes in $pl[i]$ and $MBNum(pl[i])$ be the number of middleboxes in $pl[i]$.

Based on a specific solution of *pl*, Quokka chooses an optimal or near-optimal path for each flow. The selected path is called *Minimum Delay Path* (*MDP*). *links*(*f*) and *mbs*(*f*) denote the sets of links and middleboxes that flow *f*'s MDP contains respectively. Let *FNum*(*m*) be the number of MDPs passing through middlebox *m*. The processing delay of *m* is $Delay(FNum(m))$, and the delay for link *l* is *Delay*(*l*). Let γ be the maximal acceptable flow latency. η is the threshold to limit the probability of packets exceeding the maximal delay. For example, $\eta = 1\%$ means less than 1% of the packets' latency is larger than γ. The problem can be formalized as:

$$\textbf{min.} \sum_{i=1}^{PoolNum} MBNum(pl[i]) \textbf{subjectto} \quad \forall\, i \in [1, PoolNum]:$$
$$MBNum(pl[i]) \le N \;\forall\, f \in FList:$$
$$\mathbf{P}(S_{lk} + S_{mb} > \gamma) < \eta \textbf{where} \quad S_{lk} = \sum_{l \in links(f)} Delay(l) S_{mb}$$
$$= \sum_{m \in mbs(f)} Delay(FNum(m)) \tag{9}$$

The sum of the link delays, S_{lk}, is a constant for a certain flow MDP. S_{mb} is the sum of the middlebox processing delays. The processing delay $Delay(FNum(m))$ of middlebox m is related to the number of passing MDPs. Therefore, to compute $\mathbf{P}(S_{lk} + S_{mb} > \gamma)$, we have to solve the joint distribution of the middleboxes' processing delays, which is infeasible because of the complexity.

According to Section 3, packets arrive at a given middlebox by Poisson Distribution. Let $E_m(FNum(m))$ the expected delay time of middlebox m. We then transform $\mathbf{P}(S_{lk} + S_{mb} > \gamma) < \eta$ into $S_{lk} + \sum_{m \in mbs(f)} E_m(FNum(m)) < \gamma$. We set an upper threshold Max_m for *FNum*(*m*), where Max_m is related to the status of m and γ. As $FNum(m) \le Max_m \Rightarrow E_m(FNum(m)) \le E_m(Max_m)$,

$$\sum_{l \in links(f)} Delay(l) + \sum_{m \in mbs(f)} E_m(Max_m) < \gamma \quad \Rightarrow \sum_{l \in links(f)} Delay(l)$$
$$+ \sum_{m \in mbs(f)} E_m(FNum(m)) < \gamma \tag{10}$$

Therefore, this problem can now be simplified as:

$$\textbf{min.} \sum_{i=1}^{PoolNum} MBNum(pl[i]) \textbf{subjectto} \;\forall\, i \in [1, PoolNum],$$
$$m \in MBSet(pl[i]) : MBNum(pl[i]) \le N,$$
$$FNum(m) \le Max_m \;\forall\, f \in FList : \sum_{l \in links(f)} Delay(l) + \sum_{m \in mbs(f)} E_m(Max_m) < \gamma \tag{11}$$

This is a combinatorial optimization problem. To analyze the complexity, we consider a sub-problem of MDP computation: given a middlebox placement solution, find the MDP for a given flow with the rate of $O(n)$. This sub-problem is NP-Complete when the maximal latency is polynomially large (Edmonds and Karp, 1972). Therefore, computing MDP is weak NP-Hard. The whole optimization problem itself is actually at least a weak NP-Hard problem.

4.2. Problem with different latency requirements

In the real networks, different flows of various applications may have different latency requirements. This scenario is a general version for that with the uniform latency requirement.

Let $\gamma(f)$ be the latency threshold of flow *f*. The optimization problem can be formulated as Eq. (12):

$$\textbf{min.} \sum_{i=1}^{PoolNum} MBNum(pl[i]) \textbf{subjectto} \quad \forall\, i \in [1, PoolNum]:$$
$$MBNum(pl[i]) \le N \quad \forall\, f \in FList:$$
$$\mathbf{P}(S_{lk} + S_{mb} > \gamma_f) < \eta \textbf{where} \quad S_{lk} = \sum_{l \in links(f)} Delay(l) \quad S_{mb}$$
$$= \sum_{m \in mbs(f)} Delay(FNum(m)) \tag{12}$$

This problem can be transformed into Eq. (13). The form of this expression is analogous to Eq. (11). However, there is a significant difference between them: flows are with different latency requirements, which brings additional challenges to the algorithm design. For a latency-insensitive flow *f*, the latency threshold $\gamma(f)$ can be set as a large constant in the implementation.

$$\textbf{min.}\quad \sum_{i=1}^{PoolNum} MBNum(pl[i])\,\textbf{subjectto}\ \forall\ i \in [1, PoolNum],$$
$$m \in MBSet(pl[i]): MBNum(pl[i]) \le N,$$
$$FNum(m) \le Max_m\ \forall f \in FList: \sum_{l \in links(f)} Delay(l) + \sum_{m \in mbs(f)} E_m(Max_m) < \gamma_f \tag{13}$$

5. Algorithm design

In this section, we design algorithms to solve the two problems of last section. We first design the basic algorithms for the uniform threshold problem, and then make some adjustments for the general one.

We now present the algorithm *MBSchedule*(G, *FList*, γ) as shown in Algorithm 1 to solve the above combinatorial problem. G and *FList* are the given topology and traffic list respectively. γ is the upper limit of the total latency that can be accepted for packets. In our approach, flow scheduling and MB positioning are decoupled. *MBPosition* (*KLevelVoting*) is used to find a solution of MB position, where the MBs are redundant to a certain extent. Therefore, for a relatively long interval, the solution of MB position does not need changing. The traffic flows are scheduled based on these MBs during the interval by the algorithm of *MDPSolution* (*MaskedViterbi*).

Algorithm 1. MBSchedule(G, *FList*, γ**).**

```
1:  $MBNumList.init()$
2:  while true do
3:     $pt$=MBPosition($G$, $FList$, $MBNumLst$)
4:     $solution$=MDPSolution($G$, $FList$, $pt$, $\gamma$)
5:     if $solution$ accepted then ▷ all flow latencies satisfied
6:        return $solution$
7:     end if
8:  $MBNumLst = MBNumLst.adjust(solution)$
9:     if $MBNumLst == null$ then
10:       return Error: insufficient pools
11:    end if
12: end while
```

This algorithm is efficient as a general framework for various flow patterns. Following, we will first give basic algorithms to solve the scheduling problem in enterprise networks. Then, we will adapt the basic algorithms for the scenario of data centers.

Given the topology G, the traffic list *FList* and the upper limit of the total delay γ, Algorithm *MBSchedule* first initializes the numbers of different types of MBs. Then, the first stage algorithm called *MBPosition*, is used to generate a possible placement solution of middleboxes. If no feasible result is returned, an error message is generated, which means γ is too small for the corresponding topology and traffic list. Based on the feasible placement result, the second stage algorithm of *MDPSolution* computes the optimal MDPs for all the flows. If the result of *solution* returned by *MDPSolution* cannot be accepted. In this case, the number of middleboxes should be increased. In our algorithm, we prefer to increase the number of the MB type with a higher load in *MBNumLst*.

In the following part of this section, we give the approximated optimization algorithms of the two stages. *MBPosition* is solved by a k-level voting algorithm and *MDPSolution* is solved by a masked-Viterbi algorithm.

5.1. K-level voting algorithm for MB placement

In k-level voting algorithm, we simulate k rounds voting to select the candidate pools for middlebox deployment. During voting, we maintain a score table for each <*pool*, *MBtype*> pair. In each round, all flows vote for all the <*pool*, *MBtype*> pairs according to the service chains and candidate pools. Temporary candidates are selected after each round of voting. The voting process is incremental, and new candidates are selected based on the sums of voted scores and old candidates in the past round.

Algorithm 2 shows the details. $G.plist$ is the pool list of topology G. In the algorithm, we first initialize *scores* and *candidate*, then we employ k rounds voting ($scores.vote()$) and selecting ($candidate.select()$). We optimize the *candidate* positions in each round.

Algorithm 2. KLevelVoting(G, *FList*, *MBNumList*).

```
1:  init $scores$ and $candidate$
2:  for $i \in [1, k]$ do
3:     for all $f \in FList$ do
4:        for all $p \in G.plist$ do
5:           $MB_x$=$f.chain[i]$
6:           if $i == 1$ then
7:              $PreHops = \{f.src\}$
8:           else
9:              $PreHops = candidate[f.chain[i-1]]$
10:          end if
11:          for all $pre \in PreCands$ do
12:             $scores.vote(<p, MB_x>, 1/ln(dist(pre, p))$
13:          end for
14:       end for
15:    end for
16:    $candidate.select(scores, MBNumList)$
17: end for
18: return $candidate.finalresult()$
```

Suppose the longest service chain of all flows contains k middleboxes. $\forall f \in FList$, let $f.chain$ be the service chain of f. In the first voting round, each flow f votes for the first middlebox type ($MB_x = f.chain[1]$) in its service chain. The set of potential previous hops of f is $PreHops = \{f.src\}$, where $f.src$ is the flow source. $\forall\ p \in G.plist$, f votes the pair <p, MB_x> with a score $1/ln(dist(f.src, p))$, where $dist(x, y)$ means the transmission delay of links from x to y. In the function of $scores.vote$, $1/ln(dist(f.src, p))$ is added to the score of the pair of <p, MB_x>. Then, according to the voted scores, m_x (determined by *MBNumLst*) pools are selected as candidate positions for the MB type of MB_x.

In the second round voting, *PreHops* changes to be set of the candidate pools (positions) for the first MB type in the service chain of f. Because the score is computed according according to the *dist* from the previous MB hop to the current pool. f needs to vote for its second middlebox type based on all candidate positions of the previous (first) MB type. The score for each pair <p, MB_x> will be accumulated by voted scores in the second round and candidate pools will be re-selected. The remaining $k-2$ rounds follow exactly the same principle. Finally, each type of middlebox has a score related to each pool. We generate the placement solution by choosing the top scored pools for each type of middleboxes accordingly, while the pool resource restriction (the capacity) is also considered here.

Algorithm *KLevelVoting* is in fact a greedy method. In each round, based on the previous scores and candidates, the algorithm votes and selects current candidate positions. The smaller *dist* (*pre*,*p*) is, the bigger score <p, MB_x> will get.

5.2. Masked-viterbi algorithm for MDP and feedback

MaskedViterbi is employed for MDP computation, and it has two stages: in the first stage, we compute the minimum possible delay (stored in *MinDelayList*) for each flow and reorder the flows according to *MinDelayList*; in the second stage, we find the MDP for each flow by the improved Viterbi algorithm (*impViterbi*).

Algorithm 3. MaskedViterbi(G,*FList*,*pt*,γ).

```
1:  for $f \in FList$ do ▹                    Stage 1: line 1–5
2:    $mindelay = Viterbi(G, pt)$
3:    add mindelay to MinDelayList
4:  end for
5:  rank FList by MinDelayList
6:  for $f \in FList$ do                    ▹ Stage 2: line 6–21
7:    init mask by the current loads of MBs
8:    $mb$=$ImpViterbi(G, pt, mask)$
9:    while mb is None do
10:     $mask.pop()$
11:     $mb$=$ImpViterbi(G, pt, mask)$
12:   end while
13:   $mdp.append(mb)$
14: if mdp satisfies the latency requirement of f then
15:     add mdp into solution
16:   else
17:     return solution with a feedback
18:   end if
19: end for
20: return solution
```

In the first stage, assuming that each middlebox can handle infinite flows and the processing latency is fixed, we compute the minimum possible delay for each flow. This is a multiple-stage shortest path problem, which can be solved by the *Viterbi* algorithm (Viterbi, 1967). In this problem, each type of middleboxes make up a *stage*, and the types in a service chain are combined sequentially into a multiple-stage. Here, we should find a shortest path that crosses the stages in series. *Viterbi* can solve this problem via dynamic programming. *impViterbi* is the improved *Viterbi* algorithm. Actually, it has no difference with mask except some initialized operations like removing the highly-loaded MBs in the set of mask.

The re-sorting of flows is a critical step in our algorithm. If a flow has a larger minimum possible delay, it is more difficult to find a solution for the flow. Thus, we let the flow with a larger path transmission delay choose NFs before the smaller one. Thus, from the global aspect, we can restrict the maximum delay. After obtaining the minimum possible delays, we rank the flows from long delay flows to short ones (in line 5).

In the second stage, taking the flow number restriction into account, we compute the real MDPs for the flows in the ranked order. *mask* is the set of all highly-loaded MBs during the computation and *MaskedViterbi* skips MBs in *mask* when choosing paths. In this way, we avoid adding flows to those MBs that are close to overloading. If *impViterbi* fails to find a satisfied MDP for the flow, we pop a MB with the smallest workload from *mask* for re-computation. Multiple times of re-computation by *impViterbi* may be triggered for a feasible solution. If we still cannot find the latency-bounded MDPs for all flows, a feedback will be returned to adjust the middlebox number list *MBNumList*. For each MB type MB_x in the flow's service chain, the feedback includes the workloads of MB_x, which is used in the next iteration of *MBSchedule* to adjust the MB numbers.

As formulated in Eq. (11), we maximum number of flows that MB m can handle is Max_m. When the flow number of m exceeds $0.9 \cdot Max_m$, we consider m is overloaded, and should be added to *mask*. Max_m is referred to a pre-built table determined by the joint distribution in Eq. (6) and the delay threshold γ. To avoid the complex computation of Eq. (6), instead of directly building the table, we randomly choose some engineering values in a reasonable range, check if latencies are satisfied, and insert satisfying values into table.

Considering the problem of dynamic chain in Mohammadkhan et al. (2015), MaskedViterbi algorithm can be extended to compute the path with branches (one iteration for one branch starting from the common node). Inspired by Mohammadkhan et al. (2015), we solve the potential ambiguity problem of dynamic chain by tagging the packet for each branch, which is similar to the part of handling header changing NFs in Mohammadkhan et al. (2015).

5.3. Online algorithm for dynamic flows

The above algorithms provide solutions for the static version problem. However, the traffic is dynamically changing, which makes the problem more complex.

The migration of flow states between NFs is always a significant challenge in the dynamically changing network. It may affect the delay sensitive flows because of the latency caused by the migration. However, as it is not the main concern in our paper, we only provide two methods to avoid the migration of flow states as much as possible. (1) In our implementation, we set an updating interval (for example, ten minutes). During the interval, the deployment of middleboxes will not change. The scheduling of a new flow is solved incrementally by the iterative algorithm of MaskedViterbi without impacting the existing flows. (2) In our scheme, each pool can run N middleboxes. The sates in the same pool are shared among all the running middleboxes. If one middlebox is shutdown, we prefer moving the flow states to another middbox of the same type. In this case, state migration can also be avoided.

The scheduling of a new flow is solved incrementally by the iterative algorithm of MaskedViterbi without impacting the existing flows. In our experiment on a server with an Intel Xeon 2.6 GHz dual-core processor, the computation for a new flow is less than 1 ms (without taking the communication latency into account), which means computation overhead of the algorithm itself is acceptable.

The significant contribution of our algorithm lies in the fact that it can be directly employed for the online dynamic scheduling, as it is an incremental algorithm. To avoid impacting general forwarding, in online version algorithms, we set an updating interval (for example, several minutes) during which the deployment of middlebox should be invariant, together with the determined forwarding paths. And the newly arriving flows during this updating interval can be handled gently altogether, as voting and MDP computation are done flow by flow.

5.4. Algorithms with different latency requirements

The above algorithms can solve the problem with the uniform latency requirement. However, in data centers, *latency-insensitive* flows and *latency-sensitive* flows should be scheduled with different priorities.

In the basic algorithms, flows vote the resource pools with a score of $\frac{1}{ln(dist(f.src,p))}$. In data center networks, we want short latency flows to have stronger voice. To do this, we replace the score with $\frac{1}{ln(\gamma_f)+ln(dis(f.src,p))}$. Thus, the *latency-sensitive flows* have a higher score than *latency-insensitive flows* when voting the same resource pool at the same middlebox candidate position.

At the first stage of *MaskedViterbi*, we calculate the MDP without any processing delay, and re-sort the flows by these MDP values. The larger the value is, the more likely the final latency exceeds the threshold. But for flows with different latency requirements, we should take γ_f into consideration. Let mdp_f be the value calculated in the first stage, we sort the flows by $\frac{1}{\gamma_f * mdp_f}$ (while in the original *MaskedViterbi*, we sort the flows by $\frac{1}{mdp_f}$). The flows f with smaller γ_f are placed in a earlier position in the new order. At the second stage of *MaskedViterbi*, flow f shall have more choices when choosing paths, because its priority

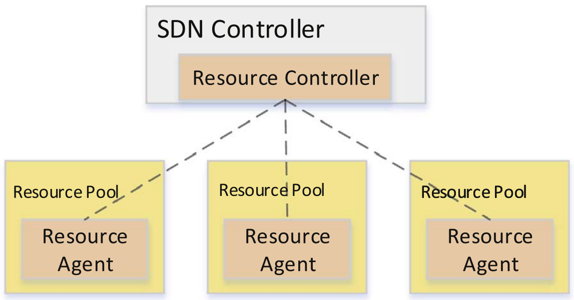

Fig. 3. Resource management framework.

is higher than the flow f' with larger $\gamma_{f'}$.

6. Implementation framework

6.1. Framework design

In the traditional cloud platforms (Zhang et al., 2014; Schwarzkopf et al., 2013; Verma et al., 2015; Hindman et al., 2011), the management of computing resource (like cpus, memories) is separated from the management of latency. However, with more and more middleboxes deployed, the latency behaviour of flows is highly tied with the resource management. To combine them together, we build a framework to manage both the computing resource (resource pools) and flow latencies. This management framework can be either built as a module in the traditional resource manager or an application in the SDN controller. Due to limited number of servers and requirement of fast resource allocation, we choose to implement our own resource manager in the SDN controller, which is shown in Fig. 3. This framework consists of a resource agent in each physical machine and a resource master. In each server, we run a resource agent, which is used to report resource utilization and allocation status periodicity to the master. We implement the resource master as an underlayer module in the SDN controller and it can be callable by other SDN controller applications.

This resource management framework mainly provides two functions: (1) resource monitor and allocation; (2) fault-handling. The resource agent reports (called *register*) the up-to-date resource states to the master when changes happen. Then, the master records the resource updates for future usage in the middlebox scheduling. After the calculation of middlebox deployment, the master sends the allocation commands to the resource agents (called *alloc*), which allocate the resource for middlebox software directly according to the commands. The process of the two actions is shown in Fig. 4.

6.2. Northbound API

Based on these functions, we provide some APIs to network operators for high-level usage.

- *addMB(flowLst, typeLst, latencyLst)*: add the minimum number of middleboxes of types in *typeLst*, to restrict the latency of flows in *flowLst* to corresponding latency in *latencyLst*. By default, *flowLst* contains all the flows in the network, and *typeLst* contains all the middlebox types. If there are no solutions, *False* will be returned.
- *delMB(flowLst, typeLst, latencyLst)*: if the middleboxes are over the provision, we may need to decrease the number of middleboxes. The parameters are the same with *addMB* above. During this process, APIs of NFV/SDN platforms like OpenNF (Gember-Jacobson et al., 2014) maybe called to merge and migrate flows.
- *optAdd(flowLst, typeLst)*: automatically add middleboxes with the types in *typeLst* for the flows in *flowLst* to achieve best-effort latency behaviour. Quokka will run *KLevelVoting-MaskedViterbi* round by round. The algorithm will be terminated when less than 5% improvement achieved in the last round.
- *minNumDeploy(flowLst, typeLst, latencyLst)*: this API calculates the minimum number of middleboxes we need to restrict delays to *latencyLst* for *flowLst*. If no solutions, *False* is returned. Otherwise, the API returns this minimum number.
- *minLatency(flowLst)*: this API returns the universal latency threshold for all the flows in *flowLst* with all available resource pools in the network.

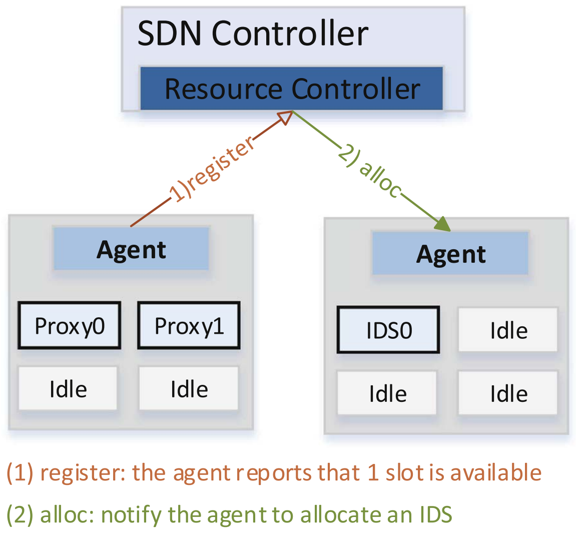
SDN Controller
Resource Controller
1)register
2) alloc
Agent
Proxy0
Proxy1
Idle
Idle
Agent
IDS0
Idle
Idle
Idle
(1) register: the agent reports that 1 slot is available
(2) alloc: notify the agent to allocate an IDS

Fig. 4. Main Actions of Resource Framework.

These APIs provide some basic high-level functions to network operators. They can be used to deploy various MB instances in available resource pools automatically.

6.3. Fault handling

To handle the faults of resource pools, we propose a two-level fault-handling mechanism. First, the master keeps the up-to-date agent states by a heart-beat protocol; second, each agent monitors the status of VMs in its resource pool. When the heart-beating between the master and the agent fails, the master marks the corresponding agent as *failed*. Then the master re-launches the middleboxes, and re-configures the network to redirect the flows to the new middleboxes. When the agent detects the failure of a VM in the resource pools, it reports the event to the master. It is the master's business to decide whether the agent should re-launch the middlebox in the original position.

We also develop a blacklist scheme to avoid frequent failures in some conditions. If the failure times of a VM exceeds a threshold, the master adds the pair *(middlebox type, resource pool)* to the blacklist. In the future, this type of middleboxes will not run on this resource pool.

7. Experiments

In this section, we evaluate the performance of our algorithms by comprehensive experiments. Based on diverse enterprise and ISP topologies, we explore the flow latency and resource utilization efficiency. During the experiments, some northbound APIs are directly used. In the following part, we will first present experiments of basic algorithms with enterprise flow pattern. And then, we will discuss experiments with different latency requirements.

Table 1
Experiment topology properties.

Name	Switch	End Node	Edge	Source
FatTree(k=16)	320	1024	4096	Custom
CERNET(2006)	41	410	116	Topo. Zoo
CARNET(2010)	44	440	86	Topo. Zoo
AS2914	70	700	222	Rocketfuel

7.1. Simulation setup

Our works focus on scheduling algorithm and should be implemented as an application in the SDN controller. Quokka can be adopted directly in real production environment, cooperating with the NFV/SDN platform like OpenNF, which provides infrastructure APIs for virtual middlebox management. To explore scheduling problems in large scale and complex networks, we have written an in-house tool for simulation usage. The project is implemented by Python, which can be found on GitHub (Quokka Project,). In the following experiments, Quokka is deployed on a controller server with an Intel Xeon 2.6 GHz dual-core processor and 4 GB ram running linux kernel 3.10.0.

We employ FatTree (Al-Fares et al., 2008), China Education and Research NETwork (CERNET) (Knight et al., 2011), Croatian Academic and Research NETwork (CARNET) (Knight et al., 2011) and Rocketfuel AS2914 (Teixeira et al., 2003) as our experiment topologies. Table 1 concludes the topology properties. We generate 10 end nodes (randomly set as resource pool or host) for each node.

We generate enterprise flows based on the real data of enterprise traffic behaviour (Nechaev et al., 2010). There are four types of flows, as shown in Table 2. Here*long* and *short* describe flow duration;*small* and *large* describe flow size. Averagely there are 2 K flows at the same time. For a specific type of flows, *Prop.* means the proportion of flow number in total flow number, while *Size* means the size proportion in summed size of all flows. *Med. Rate* is the median rate of flows.

Then, according to statistical results from Nechaev et al. (2010), we generate the traffic traces of some common network applications (e.g., ssh, NFS and Dantz), and configure the middlebox service chains referring to our campus network. To show the efficiency of Quokka, we also implement two traditional schemes for comparisons.

1. Fixed placement and fixed configuration (*Fixed-Fixed*): we deploy fixed number of middleboxes in fixed pools, and when a flow is generated, *Fixed-Fixed* sends it to the nearest middleboxes. This scheme is usually used in traditional networks to manage hardware middleboxes.
2. Load balancing with fixed placement (*Fixed-LB*): *Fixed-LB* deploys a fixed number of middleboxes and load balance the traffic among them online, analogous to Simple-fying (Qazi et al., 2013).

In the implementation of Quokka, we initialize the number of a specific MB type (MB_x) according to assumed capability of MB_x and the number of flows with MB_x. For example, if there are 1 K flows with MB_x in their service chains and each MB_x can handle 500 flows simultaneously, the initiate number of MB_x is set to be 2. The total initiate number of MBs is a multiple of 5 (the number of MB types).

Table 2
Enterprise flow categories and properties.

Dur.-Size	Prop. (%)	Size (%)	Med. Rate (bps)
Short-Small	57.2	0.6	500 K
Short-Large	2.6	0.8	10 M
Long-Small	31.8	0.8	10 K
Long-Large	8.4	97.8	100 K

After each round of *KLevelVoting-MaskedViterbi*, we add a certain number (i.e., 5 in our implementation) of MBs according to the load of each MB. We prefer to increase the number of the MB type with a higher load. For more implementation details, please refer to our project on GitBub (Quokka Project,).

7.2. Experiments results and analysis

Quokka is a scalable and efficient scheduling algorithm, as it can add or remove middlebox instances when the network load changes. Fig. 5 shows the cumulative distribution of flow latencies of Quokka with three *KLevelVoting-MaskedViterbi* iterations. During each iteration, Quokka adds new middleboxes in the positions according to *KLevelVoting* and reschedules the flows using *MaskedViterbi* Algorithm. Finally, Quokka decreases the flow latencies as the Fig. 5 shows. In (a), (b) and (c), the curve between 40 and 60 ms is because of the topology property, especially the transmission delays between the sources and destinations. These three topologies have the similar features in this aspect.

This experiment shows that, Quokka can dynamically change the number of middleboxes with respect to processing workloads. The number of middleboxes changed in each iteration is simply calculated with traffic requirements. In our experiments, the iteration process converges very quickly. For most topologies, three iterations are enough. And this makes Quokka very computing-friendly.

With the same number of middleboxes deployed, we apply *Fixed-Fixed*, *Fixed-LB* and Quokka in the controller respectively. As shown in Fig. 6, Quokka has smaller flow latencies. In AS2914, *Fixed-LB* works even worse than *Fixed-Fixed*, as AS2914 contains links with large latencies. In this type of topology, link transmission delay mostly contributes much more to the total latency than middlebox processing. This is a problem existing in the traditional load balancing. Quokka avoids this problem, thus is always very stable in diverse topologies and achieves overwhelming performance.

It is seen from this experiment that, Quokka receives better latency performance over both nearest choice (*Fixed-Fixed*) and simple load balancing (*Fixed-LB*). *Fixed-Fixed* only chooses the nearest middleboxes, and it makes some instances overloaded. As overloaded middleboxes need longer time to process packets, the flows across it may be delayed. Even though *Fixed-LB* utilizes load balancing technologies to avoid *overheated* situations, it may lead flows to faraway instances. Instead, Quokka seeks the balance between *nearest position* and middlebox *burden* when scheduling flows. Alternately as shown in Fig. 6, Quokka is very efficient in shortening long-tailed flows, which is significant for delay guarantee.

Furthermore, we calculate latency reduction ratio Quokka achieves over other two schemes. We first calculate the average latencies of all flows during the experiment, and then get the ratio reduced by Quokka over *Fixed-Fixed* and *Fixed-LB*. The results are shown in Table 3. From the experiment, we see that Quokka reduces about 20% of latency over the traditional schemes on average.

In the first three topologies, *Fixed-Fixed* works even bad, as it has the disadvantage of at least 20% compared to both simple load balancing and Quokka. It is because the nearest choice method makes many middleboxes overhead, and thus performs poorly. Still, in the first three topologies, Quokka performs 4–8% better than *Fixed-LB*. However, in AS2914, the performances of *Fixed-Fixed* and *Fixed-LB* are reversed. As explained above, AS2914 is a complex ISP networks across various countries and cities, and it has many long latency links. For such reason, simple load balancing may bring extra delays similar to the toy example in Fig. 1. In this topology, up to 43% of performance improvement is achieved by Quokka, compared to traditional load balancing.

When given a max delay restriction, we apply these schemes to provide delay guarantees. For each algorithm and each topology, we carry out our experiment for 5 times. If the delays are restricted for at

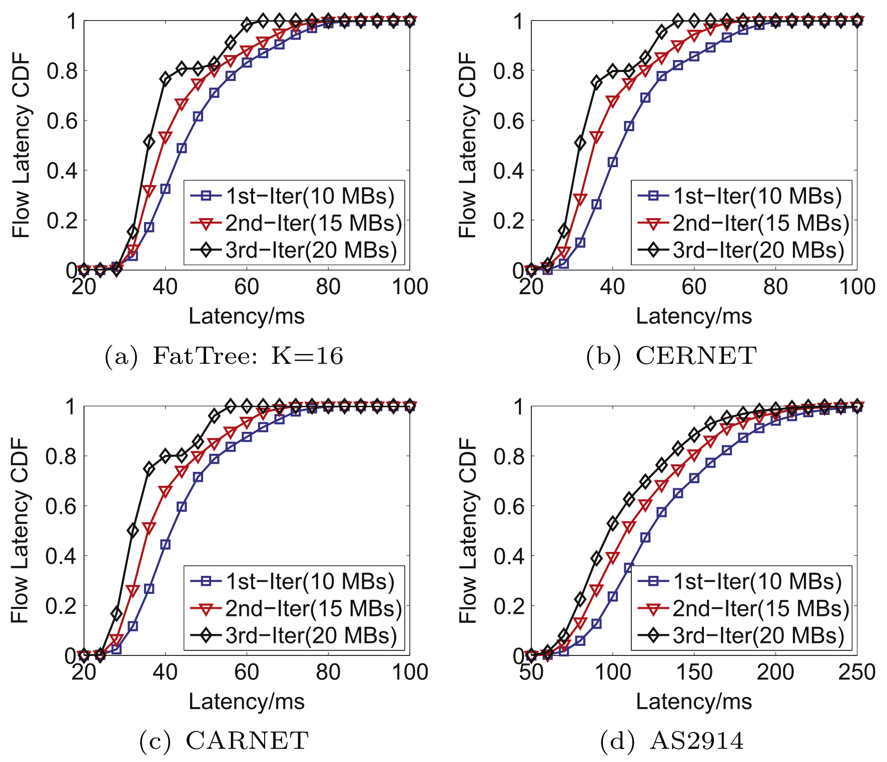

Fig. 5. Latency Distributions of Quokka with Three Iterations. (a) FatTree: K=16 (b) CERNET (c) CARNET (d) AS2914.

least 4 times, we regard the delay is guaranteed. Otherwise, we think the maximum delay threshold is violated by the corresponding algorithm. In fact, only *Fixed-Fixed* method cannot guarantee the 120 ms delay in FatTree(k=16), and this is denoted as a * symbol in Table 4. Other experiments are all successful in restricting delays for 5 times. In most cases, Quokka reduces the resource usage by 30–50%, compared to other two schemes.

Table 3
Delay reduced ratio .

	FatTree	CERNET	CARNET	AS2914
Quokka/Fixed-Fixed	35.0%	35.1%	25.8%	4.28%
Quokka/Fixed-LB	4.28%	8.36%	8.49%	43.3%

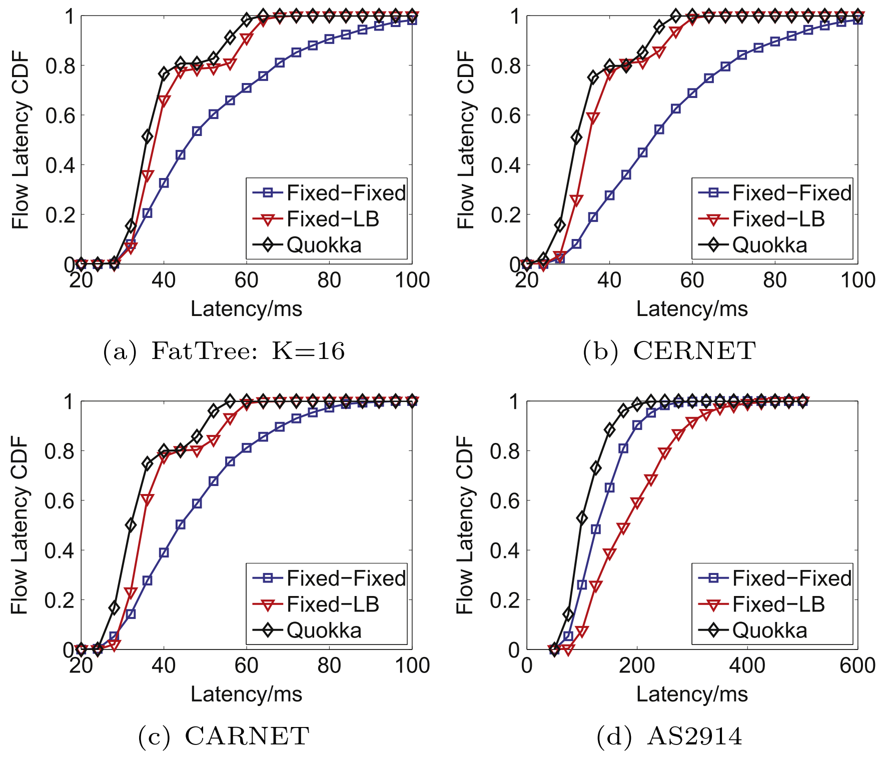

Fig. 6. Latency Distributions of Three Algorithms(20MBs). (a) FatTree: K=16 (b) CERNET (c) CARNET (d) AS2914.

Table 4
MB numbers needed to restrict delay.

Topo.	Delay (ms)	Fixed-Fixed	Fixed-LB	Quokka
FatTree	120	*	15	10
CERNET	110	20	15	10
CARNET	90	15	10	10
AS2914	450	20	15	10

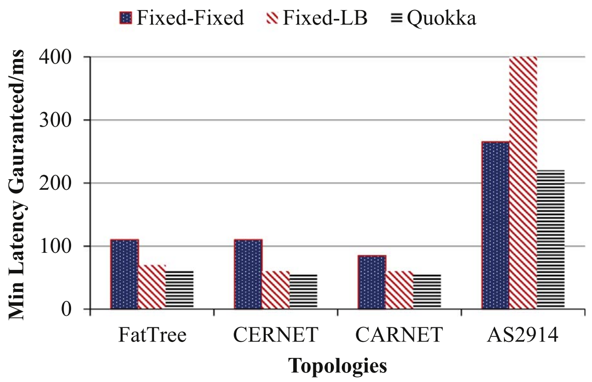

Fig. 7. Minimum delay guaranteed by algorithms.

Given the same number of middleboxes, we then explore the minimum possible delay each algorithm can provide. In the following experiments, we evaluate these three algorithms, and find Quokka can always provide smaller possible delay restriction than *Fixed-Fixed* and *Fixed-LB* in diverse topologies. As shown in Fig. 7, in the first three topologies, Quokka gets about 40% smaller delay threshold than *Fixed-Fixed*, and slight smaller than *Fixed-LB*. In AS2914, Quokka achieves obvious better latency performance than both static configuration and simple load balancing.

The *Long-Large* flows consist of most of packets (97.8% in size), as shown in Table 2, so setting an updating interval does not lead to serious optimality drawbacks in the online version algorithm. Experiments show that, static version algorithm and online version achieve similar performances.

7.3. Experiments with different latency requirements

For the growing deployment of middleboxes in a wide range of areas, we would like to extend our work for various flow patterns. In this subsection, we do some experiments with different latency requirements.

The flows in our experiments are based on the data center tracing from Google Inc. and statistical results from Wilson et al. (2011), Verma et al. (2015), Benson et al. (2010), Kandula et al. (2009). As the tracing lacks middlebox information, we configure the flow service chains according to their types (Benson et al., 2010). The latency thresholds are determined by the observation experiences in Wilson et al. (2011), Vamanan et al. (2012).

We have three type of flows according to their latency requirements: (1) *real time flows (50%)*, which are usually critical user-reactive requirements, and should be delivered in 150 ms; (2) *soft real time flows (10%)*, typical flows of this type are background refreshing when we are surfing the websites. Their requirements are not very critical as the first type, but still should be processed in 300 ms; (3) *none real time flows (40%)*, this kind of flows are usually long-time running backup jobs or logging jobs. We set a threshold of 1000 ms for this kind of flows. The distribution of the flows follow the statistical results of Benson et al. (2010).

Given the same resource, we evaluate our algorithm (called *Quokka-DC*) together with *Fixed-Fixed* and *Fixed-LB* to see whether the latencies are guaranteed. We carry out several rounds of experiments, and find *Quokka-DC* can greatly reduce the deadline mismatch ratio. As shown in Table 5, *Quokka-DC* can always have less deadline mismatch ratio than the other two algorithms in all these four topologies. From Table 5, we see that, in most of times, *Quokka-DC* only have less than one tenth mismatch ratio of that in *Fixed-Fixed* or *Fixed-LB*. Only several flows cannot meet their completion times with *Quokka-DC*. Especially in AS2914, both of the traditional methods have a poor performance. This shows *Quokka-DC* can still work very well in complex topologies, while simple load balancing and static configuration cannot meet various topologies.

Table 5
Deadline mismatch ratio.

Topo.	Fixed-Fixed (%)	Fixed-LB (%)	Quokka-DC (%)
FatTree	7.96	2.80	0.02
CERNET	15.8	11.4	1.61
CARNET	25.3	12.2	0.00
AS2914	23.9	30.5	1.61

Like the measurement benchmark in enterprise networks, if an algorithm can meet 99% more flow deadlines in 4 out of 5 experiments, we think this algorithm can provide guarantees for this topology. In our experiments, all of these three algorithms cannot provide guarantees for AS2914, because their are no solutions for some flows. And in other three topologies, *Quokka-DC* can provide guarantees within 20 MBs with about 2 K flows, while neither *Fixed-Fixed* or *Fixed-LB* can guarantee the deadlines.

In the experiments of enterprise networks, simple load balancing approach sometimes provides a convincing performance. But in the experiments for data center flow patterns, simple load balancing always works poorly. Because many flows in data center pattern have more critical deadlines than the flows in enterprise pattern. Data center flows are generally more latency-sensitive than enterprise flows.

To show the flow latency distribution clearly, we ignore *none real time flows*, and present *real time flows* and *soft real time flows* separately. Fig. 8 shows the latency distribution CDF of *real time flows*. Almost in all times, the CDF curve of *Quokka-DC* is above the other algorithms, except 120 ms in FatTree. At that point, *Fixed-LB* works slightly better. It may seem that *Quokka-DC* is not good enough at first glance. However, the latency threshold of *real time flows* is 150 ms, and 99.98% flows in FatTree can be restricted within this threshold using *Quokka-DC*. While, only 97.2% flows can be guaranteed using *Fixed-LB*. Simple load balancing employs some greedy method, while Quokka can make a good plan for all the flows.

In CERNET and CARNET, at the beginning of x-axis, *Fixed-Fixed* works better than *Fixed-LB*. However, after a specific point, *Fixed-LB* begins to overcome *Fixed-Fixed*. As static configuration sends packets to their nearest middleboxes, some flows can get good latency performance. But there are much more flows sent to overloaded instances and the total performance is very bad. Load balancing can relieve this problem somewhat, but still not very good. In complex topologies like AS2914, as we explained in enterprise networks, *Fixed-LB* works even worse than *Fixed-Fixed*. On the other hand, Quokka can always get noticeable performance improvement in all the four topologies.

The distribution of *soft real time flows* is shown in Fig. 9. This figure is similar to Fig. 8, except that the thresholds are different. If we combine Figs. 8 and 9 together, we find our algorithm treats different type of flows with different priorities. *Quokka-DC* tries its best to provide according latency guarantees to different flows with limited resource. Experiments show that Quokka can make a good balance of latency requirements of various flows and limited computing resource.

According to our definition, there is only a universal threshold in enterprise networks, while there are multiple thresholds in data centers. With a single threshold, the algorithm just needs to allocate

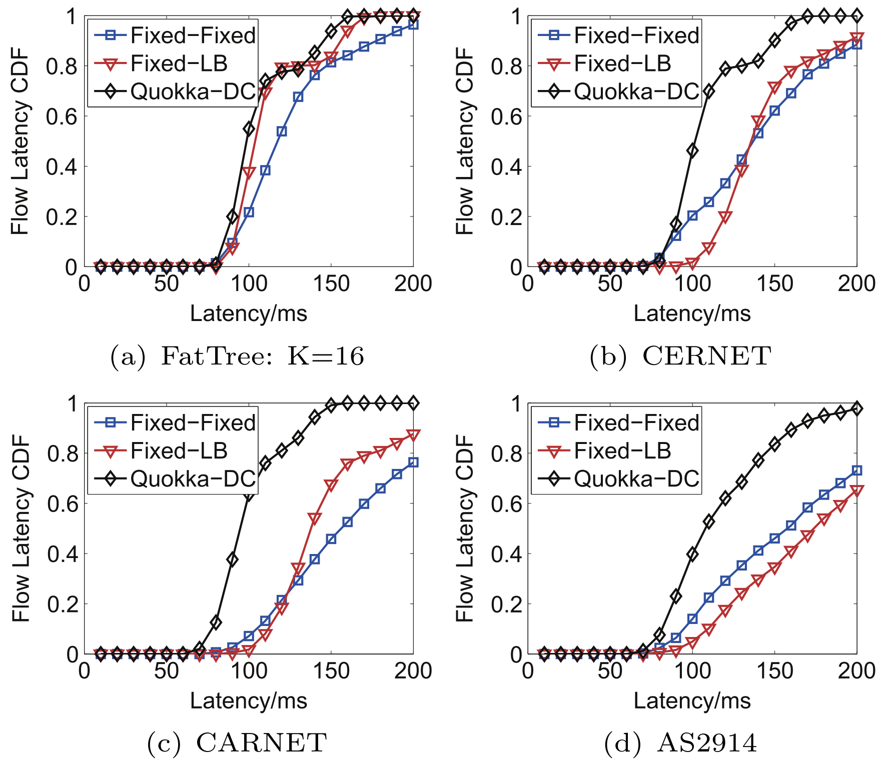

Fig. 8. Latency distributions of real time flows (20 MBs). (a) FatTree: K=16 (b) CERNET (c) CARNET (d) AS2914.

the resource equally to the flows. *Fixed-LB* is also a equal algorithm. However, some flows are very probable to have a longer latency, and Quokka gives these flows higher priorities to avoid probable deadline mismatch. When handling multiple thresholds, latency thresholds should be considered altogether when choosing paths for different flows. These are in fact what Quokka does compared to simple load balancing and static configuration.

Even though enterprise flow pattern and data center flow pattern are different, they share the same algorithm framework in Quokka. This general algorithm framework can be used for diverse flow patterns with different requirements of latency. In our experiments, we only change serval lines of code when migrating the basic algorithms to *Quokka-DC*. We believe our algorithms can be deployed in different environments easily.

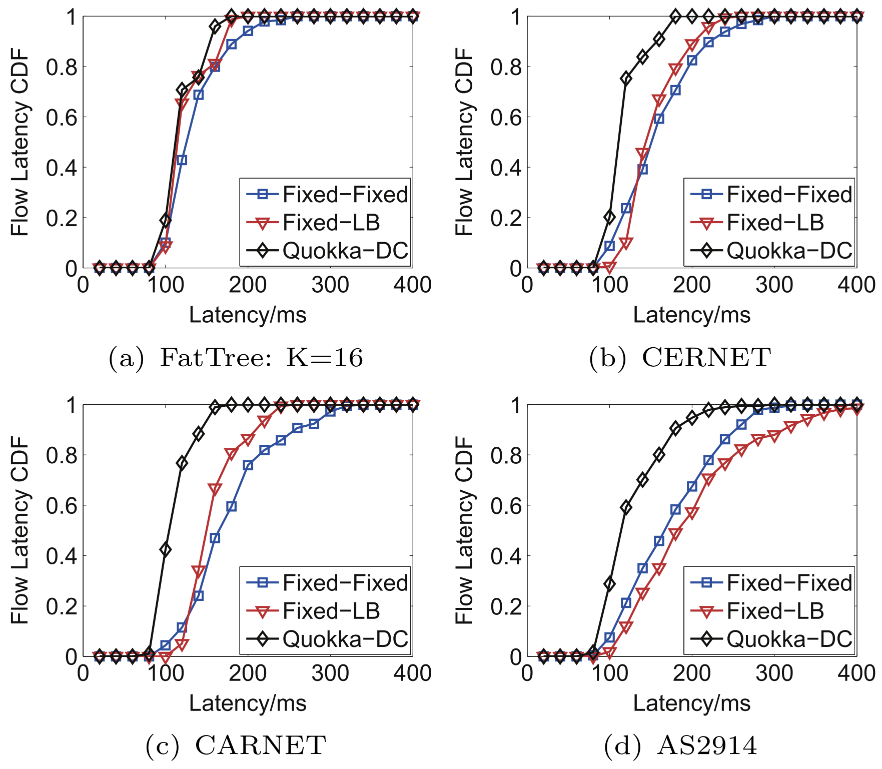

Fig. 9. Latency distributions of soft real time flows (20 MBs). (a) FatTree: K=16 (b) CERNET (c) CARNET (d) AS2914.

7.4. Computing complexity

In our scheduling algorithm, loops of *KLevelVoting-MaskedViterbi* often iterate at most 3 times. And in Algorithm *KLevelVoting*, the number of global loops is $|k|\cdot|FList|\cdot|G.\ plist|$. $|k|$ is the length of longest service chain and $|G.\ plist|$ is the number of resource pools. In our experiments, $|k|$ is smaller than 10, and $|G.\ plist|$ is about 200 at most. Due to this, the number of loops is linear with respect to the number of flows $|FList|$.

In each loop of *KLevelVoting*, we should calculate the shortest paths between all switches. In our implementation, *Floyd-Warshall Algorithm* (Floyd, 1962) is adopted to handle it. Its complexity of $O(|V|^3)$ where $|V|$ is number of nodes in topology, and experiments show this part contributes most to the computing complexity. However, topologies of switches are stationary during the whole experiments, and the shortest paths between pairs can be calculated in advance. Even when resource pools (terminal nodes) change, the topologies of switches keep stationary. Thus each loop in *KLevelVoting* executes in constant time.

And in Algorithm *MaskedViterbi*, the number of global loops is also linear with respect to flow numbers (as analyzed above, $|f.\ chain|$ should be a constant; $|mask|$ is a small set of resource pools, and it is also a small integer). And during each loop, *Viterbi Algorithm* is executed once. As we mentioned above, *Viterbi Algorithm* is used to solve a multiple-stage shortest path problem. However, in our experiments, the number of stages and the number of middleboxes in each stage are both small integers, so the multiple-stage shortest path problem can be solved in constant time. Also, the complexity of each loop in *MaskedViterbi* is constant time.

Thus, the computing complexity of our algorithm is linear with respect to flow numbers theoretically. And experiments show that Quokka is very friendly in resource consuming, as the codes add no obvious overhead in experiments. In dynamic version algorithm, the consuming is even less, since the algorithm runs only once during an updating interval (often several minutes).

7.5. Optimization for sparse networks

As described above, the computing complex is proportional to the flow number. When the topology is sparse and flow number is large, the algorithm is not reasonable. We make some optimizations for sparse networks.[2]

The main technology we adopt is flow merging. Instead of voting and choosing path by each flow, flows with similar properties are first merged as *super flows*. Each super flow has a magnification factor, which is in fact the real flow number included in the super flow. And when scheduling we use the real flow number to vote and schedule flows. Although the computing complexity is still $O(|FList|)$, the voting and scheduling processes are highly reduced.

8. Related work

8.1. Middlebox management

Quokka benefits from many previous works. Simple-fying (Qazi et al., 2013) and FlowTags (Fayazbakhsh et al., 2014) enforce middlebox-related policies with SDN technologies, and they simply headache middlebox configurations gracefully. Configurations of middleboxes in traditional networks are complex, and some policies may be difficult to deploy. Simple-fying and FlowTags extend southbound interface of SDN to manage middleboxes together with Openflow (McKeown et al., 2008) switches. They distinguish different states of packets using IP header fields, and manage both middlebox states and forwarding states uniformly in the controller. Their contributions mostly lie in middlebox configurations.

OpenNF (Gember-Jacobson et al., 2014) embraces SDN with NFV technology, and provides rich APIs of middlebox operations for high layer applications. Utilizing these interfaces, we can easily copy and migrate stateful/stateless middleboxes. OpenNF also provides some options like loss-free and keep-order when copying or immigrating middleboxes, thus network managers themselves can balance operation speed with operation qualities. OpenNF can be used as the underlayer libraries when Quokka is employed in real production environment. But OpenNF itself doesn't provide any scheduling algorithm. Quokka is based on such NFV/SDN platform, and contributes to high level scheduling algorithms.

Previous projects like Click (Kohler et al., 2000) and RouteBricks (Dobrescu et al., 2009) are devoted to immigrating network functions from hardware ASICs to software platforms. Click provides programmability and modularity for future NFV platforms such as CoMB (Sekar et al., 2012), NetVM (Hwang et al., 2014) and xOMB (Anderson et al., 2012). RouteBricks tries to build a scaling software router and demonstrate a 35 Gbps parallel router prototype. CoMB, NetVM and xOMB focus on building high performance network function platforms. NetVM is built on commodity servers and optimizes inter-VM communication together with CPU scheduling. It improves both network throughput and network processing speed, and can be customized as firewalls, proxies and routers. xOMB provides an incrementally scalable platform for software middleboxes, and demonstrates good performance for load balancing, protocol acceleration and application integration. Alternately, xOMB's internal structure is used as a typical model to analyze middlebox latency behaviours.

Static policies are often used in hardware middlebox configuration (Sherry et al., 2012), but they lack flexibility and scalability. A traditional load balancing example is introduced by Simple-fying (Qazi et al., 2013). However, Simple-fying only sets maximum flow number thresholds for middleboxes, and doesn't pay much attention to packet latencies, while Quokka achieves high performance on reducing end-to-end delays.

In Mohammadkhan et al. (2015), Rankothge et al. (2015), the authors proposed the optimization problem to solve the NF (middlebox) placement and traffic scheduling together. The heuristic The network Function Center (NFC) (Rankothge et al., 2015) is a good architecture to solve the problem. However, in the paper, they mainly focused on the resource manager (one of the five modules of NFC) and did not provide the design details of flow manager, elasticity manager, etc.

8.2. NFV

ClickOS (Martins et al., 2014) builds a tiny VM image for middlebox software, and it makes realtime launching possible for middleboxes. ClickOS concentrates on high performance middlebox VMs, which is relied on by future scheduling platforms. ClickOS boots fast (about 30 ms), consumes less resource (image size is only 5MB) and achieves high networking throughput (up to 30 Gb/s). Specially, ClickOS is based on modular router Click (Kohler et al., 2000), and also benefits much from previous high performance network function platforms. The work of Abdou et al. (2015) employs a simple queueing model when exploring software middlebox latency behaviours. On the basis of this simple model, we further explore more complex situations of software middlebox.

8.3. Latency management

Many resource management frameworks are built in prior clouds (Zhang et al., 2014; Schwarzkopf et al., 2013; Verma et al., 2015; Hindman et al., 2011) to manage millions of servers in data centers. These frameworks pay more attention to scalability. However, the

[2] We define **sparse** by the **node number/flow number** ratio.

operations like resource allocation take several seconds to complete. In Liu et al. (2016), they propose *Footprint* to guarantee the QoS of online services in the "integrated" setting of proxies, data centers, and the wide area network. The latency can be efficiently controlled in SDN-based integrated way. We build our own framework in the SDN controller to make resource management faster, and during this process, Quokka benefits much from these works.

Prior works of latency management in data center network like DCTCP (Alizadeh et al., 2010), D^{2}TCP (Vamanan et al., 2012) are some adaptable version of TCP in data centers. They modify the sliding window method of TCP to support different latency priorities. And switching mechanisms are also changed to forward packets according to their deadlines. We follow a different approach in this paper, as we combine management of latency and middlebox scheduling, and we guarantee different latencies by choosing proper paths and middleboxes.

9. Conclusion and future work

As diverse middleboxes are adopted in various networks, ie. enterprise networks and data center networks, the management of middleboxes is becoming a great challenge for network operators, especially taking latency into consideration. Latency is a critical property in many networks, and the management of latency is unfortunately ignored in many management framework. In this paper, motivated by NFV technologies, we combine the management of middleboxes and latency together.

In this paper, we explore software middlebox latency behaviours, and present Quokka, a dynamic middlebox scheduling scheme based on NFV/SDN platforms. Compared with traditional schemes, Quokka deploys middleboxes according to realtime traffic, and provides lower latencies for end users with the same resource.

To handle universal-threshold flows, we build a basic scheduling framework to provide latency guarantees for flows. Later, we extend our algorithms for more complex and fine-grained latency control in data centers. Experiments show that our algorithm framework can be easily modified for various flow patterns, while simple load balancing and static configuration only work in very simple topologies and their performance are always poor than our algorithms, especially in complex scenarios.

In the future, we plan to implement Quokka in real systems. Although many NFV/SDN platforms like OpenNF can be used when deploying Quokka in testbed experiments, many engineering problems should be solved, for instance, eliminating flow impacting. Additionally, how to implement the multiple NFVs in a multi-core server is also significantly important (Savi et al., 2015). For example, sharing a core between two different NFVs may cause the overhead of non-uniform memory access (NUMA) (Mohammadkhan et al., 2015).

Acknowledgement

We are grateful to Peihan Miao and Christos Papadimitriou from Berkeley for their help in complexity analysis of the optimization problem. This work is supported by the National Research Program of China (973) under Grant no. 2012CB315803, the National Natural Science Foundation of China under Grant no. 61402255, the R & D Program of Shenzhen under Grant no. ZDSYS20140509172959989, No. JCYJ20150630170146830, and No. Shenfagai(2015)986.

References

Abdou, A., Matrawy, A., van Oorschot, P.C., 2015. Taxing the queue: hindering middleboxes from unauthorized large-scale traffic relaying. IEEE Commun. Lett. 19 (1), 42–45.
Adan, I., Resing, J., 2002. Queueing Theory. Eindhoven University of Technology Eindhoven, Eindhoven, Netherlands.
Al-Fares, M., Loukissas, A., Vahdat, A., 2008. A scalable, commodity data center network architecture. In: Proceedings of the ACM SIGCOMM, Seattle, USA.
Alizadeh, M., Greenberg, A., Maltz, D.A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., Sridharan, M., 2010. Data center tcp (dctcp). In: Proceedings of the ACM SIGCOMM, New Delhi, India.
Allen, A.O., 1978. Probability, Statistics, and Queueing Theory with Computer Science Applications. Academic Press, Waltham, USA.
Amazon EC2 Website. ⟨http://aws.amazon.com/ec2⟩.
Anderson, J.W., Braud, R., Kapoor, R., Porter, G., Vahdat, A., 2012. xomb: Extensible open middleboxes with commodity servers. In: Proceedings of ACM/IEEE ANCS, Austin,USA.
Benson, T., Akella, A., Maltz, D.A., 2010. Network traffic characteristics of data centers in the wild. In: Proceedings of the ACM SIGCOMM IMC, Melbourne, Australia.
Casado, M., Freedman, M.J., Pettit, J., Luo, J., McKeown, N., Shenker, S., 2007. Ethane: Taking control of the enterprise. In: Proceedings of the ACM SIGCOMM, Kyoto, Japan.
Chowdhury, N.M.K., Boutaba, R., 2010. A survey of network virtualization. Comput. Netw. 54 (5), 862–876.
Dobrescu, M., Egi, N., Argyraki, K., Chun, B.-G., Fall, K., Iannaccone, G., Knies, A., Manesh, M., Ratnasamy, S., 2009. Routebricks: exploiting parallelism to scale software routers. In: Proceedings of the ACM SIGOPS, Big Sky, USA.
Edmonds, J., Karp, R.M., 1972. Theoretical improvements in algorithmic efficiency for network flow problems. J. ACM 19 (2), 248–264.
Fayazbakhsh, S.K., Chiang, L., Sekar, V., Yu, M., Mogul, J.C., 2014. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In: Proceedings of USENIX NSDI, Seattle, USA.
Floyd, R.W., 1962. Algorithm 97: shortest path. Commun. ACM 5 (6), 345–348.
Gember, A., Krishnamurthy, A., John, S.S., Grandl, R., Gao, X., Anand, A., Benson, T., Akella, A., Sekar, V. Stratos: A network-aware orchestration layer for middleboxes in the cloud, CoRR abs/1305.0209.
Gember-Jacobson, A., Viswanathan, R., Prakash, C., Grandl, R., Khalid, J., Das, S., Akella, A., 2014. Opennf: Enabling innovation in network function control. In: Proceedings of ACM SIGCOMM, Chicago, USA.
Ghodsi, A., Sekar, V., Zaharia, M., Stoica, I., 2012. Multi-resource fair queueing for packet processing. In: Proceedings of the ACM SIGCOMM, Helsinki, Finland.
Google Inc., ⟨https://github.com/google/cluster-data⟩.
Gross, D., 2008. Fundamentals of Queueing Theory. John Wiley & Sons, Hoboken, USA.
Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A.D., Katz, R.H., Shenker, S., Stoica, I., 2011. Mesos: A platform for fine-grained resource sharing in the data center. In: Proceedigns of the Usenix NSDI, Boston, USA.
Hwang, J., Ramakrishnan, K.K., Wood, T., 2014. Netvm: High performance and flexible networking using virtualization on commodity platforms. In: Proceedings of the USENIX NSDI, Seattle, USA.
Joseph, D.A., Tavakoli, A., Stoica, I., 2008. A policy-aware switching layer for data centers. In: Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication, Seattle, WA, USA.
Kandula, S., Sengupta, S., Greenberg, A., Patel, P., Chaiken, R., 2009. The nature of data center traffic: measurements & analysis. In: Proceedings of the ACM IMC, Chicago, USA.
Kim, H., Feamster, N., 2013. Improving network management with software defined networking. IEEE Commun. Mag. 51 (2), 114–119.
Knight, S., Nguyen, H.X., Falkner, N., Bowden, R., Roughan, M., 2011. The internet topology zoo. IEEE JSAC 29 (9), 1765–1775.
Kohler, E., Morris, R., Chen, B., Jannotti, J., Kaashoek, M.F., 2000. The click modular router. ACM Trans. Comput. Syst. 18 (3), 263–297.
Liu, H.H., Viswanathan, R., Calder, M., Akella, A., Mahajan, R., Padhye, J., Zhang, M., 2016. Efficiently delivering online services over integrated infrastructure. In: Proceedings of the USENIX NSDI, Santa Clara, USA.
Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., Huici, F., 2014. Clickos and the art of network function virtualization. In: Proceedings of the USENIX NSDI, Seattle, USA.
McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J., 2008. Openflow: enabling innovation in campus networks. ACM SIGCOMM Comput. Commun. Rev. 38 (2), 69–74.
Mohammadkhan, A., Ghapani, S., Liu, G., Zhang, W., Ramakrishnan, K.K., Wood, T., 2015. Virtual function placement and traffic steering in flexible and dynamic software defined networks. In: Proceedings of the IEEE LANMAN, Beijing, China.
Mohammadkhan, A., Liu, G., Zhang, W., Ramakrishnan, K.K., Woodv, T., 2015. Protocols to support autonomy and control for nfv in software defined networks. In: Proceedings of the IEEE NFV-SDN, San Francisco, USA.
Munir, A., Qazi, I.A., Uzmi, Z.A., Mushtaq, A., Ismail, S.N., Iqbal, M.S., Khan, B., 2013. Minimizing flow completion times in data centers. In: Proceedings of the IEEE INFOCOM, Turin, Italy.
Nechaev, B., Allman, M., Paxson, V., Gurtov, A., 2010. A preliminary analysis of tcp performance in an enterprise network. In: Proceedings of the INM/WREN, San Jose, USA.
Nunes, B., Mendonca, M., Nguyen, X.-N., Obraczka, K., Turletti, T., 2014. A survey of software-defined networking: past, present, and future of programmable networks. IEEE Commun. Surv. Tutor. 16 (3), 1617–1634.
Qazi, Z.A., Tu, C.-C., Chiang, L., Miao, R., Sekar, V., Yu, M., 2013. Simple-fying middlebox policy enforcement using sdn. In: Proceedings of the ACM SIGCOMM, Hong Kong.
Quinn, P., Guichard, J., Kumar, S., Agarwal, P., Manur, R., Chauhan, A., Leymann, N., Leymann, N., Boucadair, M., Jacquenet, C., Smith, N., Yadav, N., Nadeau, T., Gray, K., McConnell, B., Glavin, K., 2014. Network service chaining problem statement. IETF Draft.
Quokka Project. ⟨https://github.com/bestdpf/pyquokka⟩.

Rankothge, W., Ma, J., Le, F., Russo, A., Lobo, J., 2015. Towards making network function virtualization a cloud computing service. In: Proceedings of the IFIP/IEEE IM, Ottawa, Canada.

Savi, M., Tornatore, M., Verticale, G., 2015. Impact of processing costs on service chain placement in network functions virtualization. In: Proceedings of the IEEE NFV-SDN, San Francisco, USA.

Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M. Wilkes, J., 2013. Omega: flexible, scalable schedulers for large compute clusters. In: Proceedings of the ACM EuroSys, Prague, Czech.

Sekar, V., Egi, N., Ratnasamy, S., Reiter, M.K., Shi, G., 2012. Design and implementation of a consolidated middlebox architecture. In: Proceedings of the USENIX NSDI, San Jose, USA.

Sherry, J., Hasan, S., Scott, C., Krishnamurthy, A., Ratnasamy, S., Sekar, V., 2012. Making middleboxes someone else's problem: Network processing as a cloud service. In: Proceedings of the ACM SIGCOMM, Helsinki, Finland.

Teixeira, R., Marzullo, K., Savage, S., Voelker, G.M., 2003. Characterizing and measuring path diversity of internet topologies. In: Proceedings of the ACM SIGMETRICS, San Diego, USA.

Vamanan, B., Hasan, J., Vijaykumar, T., 2012. Deadline-aware datacenter tcp (d2tcp). In: Proceedings of the ACM SIGCOMM, Helsinki, Finland.

Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., Wilkes, J., 2015. Large-scale cluster management at google with borg. In: Proceedings of the ACM EuroSys, Bordeaux, France.

Viterbi, A.J., 1967. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. IEEE Trans. Inf. Theory 13 (2), 260–269.

Wilson, C., Ballani, H., Karagiannis, T., Rowtron, A., 2011. Better never than late: Meeting deadlines in datacenter networks. In: Proceedings of the ACM SIGCOMM, Toronto, Canada.

Zhang, Z., Li, C., Tao, Y., Yang, R., Tang, H., Xu, J., 2014. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. In: Proceedings of the VLDB, Hangzhou, China.