TRIPOD: Towards a Scalable, Efficient and Resilient Cloud Gateway

Menghao Zhang, Jun Bi, *Senior Member, IEEE*, Kai Gao, Yi Qiao, Guanyu Li, Xiao Kong, Zhaogeng Li, and Hongxin Hu, *Member, IEEE*

***Abstract*— Cloud gateways are fundamental components of a cloud platform, where various network functions (e.g., L4/L7 load balancing, network address translation, stateful firewall, and SYN proxy) are deployed to process millions of connections and billions of packets. Providing *high-performance* and *failure-resilient* packet processing with a *scalable traffic management* mechanism is crucial to ensuring the quality of service of a cloud provider, and hence is of great importance. Many network functions nowadays are implemented in software with commodity servers for low cost and high flexibility. However, existing software-based network function frameworks oftentimes provide part of these features, while cannot satisfy all three requirements above simultaneously. To address these issues, in this paper, we introduce TRIPOD, a novel network function framework specialized for cloud gateways. Having identified the fundamental limitations of loosely coupling *traffic*, *processing logic* and *state*, TRIPOD jointly manages these three elements with the unique characteristics of cloud gateways, which is enabled by a *simple, efficient traffic processing mechanism*, and a *high performance state management service*. Adopting several effective techniques and optimizations, TRIPOD is able to achieve *scalable traffic management* (<100 flow rules for even ∼Tbps traffic), *high performance* (reducing 40% of latency compared with state of the art) and *failure resilience* (similar packet/connection loss rate compared to state of the art), with reasonable overheads (less than 10% of the workload traffic) even under an extremely heavy traffic, making it a good fit for cloud gateways.**

***Index Terms*— Cloud gateway, scalable traffic management, high performance, failure resilience.**

I. INTRODUCTION

LARGE-SCALE commercial cloud platforms [1]–[5] have become a crucial part of today's Internet, hosting millions of services and serving billions of clients all over the world. Controlling the traffic between a data center and the Internet or other data centers, cloud gateways play an important role in providing an efficient, reliable networking service. A typical cloud gateway has multiple network functions (NFs), such as L4/L7 load balancing, network address translation, stateful firewall, and SYN proxy, which are essential to ensure security, improve performance, and provide other novel network functionalities.

To enable more flexible and low-cost packet processing, there's a growing trend to *softwarize* network functions on commodity servers instead of dedicated proprietary hardware. The general form of such an approach, often referred to as *Network Function Virtualization* (NFV),[1] has been discussed in a broader context where various frameworks are proposed to achieve high performance [6]–[9], failure resilience [10]–[12] and ease of management [13]–[16].

While previous studies [17]–[20] have achieved softwarized load balancing, it is still unknown whether more general network functions at the cloud gateway can take advantage of these existing NFV frameworks to achieve high scalability, high performance and high availability. In particular, a cloud gateway framework must **satisfy the following requirements simultaneously**:

- **R1: Scalable traffic management:** Cloud gateways have an extremely large address space, traffic volume with numerous concurrent connections [21]–[23], making SDN-based fine-grained traffic steering and flow rule updating a challenging problem. A cloud gateway framework must be combined with a ***scalable*** traffic management mechanism to direct the traffic to the corresponding NF instance correctly and timely.
- **R2: High performance:** Cloud gateways have a very high standard for throughput and latency, usually at the speed of ∼Tbps and the latency of tens of microseconds (μs) [24], where virtualization is always not the first preference. A cloud gateway framework must be able to achieve ***efficient*** packet processing.
- **R3: Low-cost failure resilience:** Cloud gateways have a high demand for failure resilience, since machine failures occur frequently [25], [26]. Otherwise, stateful network functions are not able to process affected TCP connections when an NF instance *fails and stops* [27], which might lead to a significant service degradation. The scale

Manuscript received October 10, 2018; revised December 27, 2018; accepted January 11, 2019. Date of publication February 5, 2019; date of current version February 14, 2019. This work was supported in part by the National Key R&D Program of China under Grant 2017YFB0801701, and in part by the National Science Foundation of China under Grant 61872426. *(Corresponding authors: Jun Bi; Kai Gao.)*

M. Zhang, J. Bi, Y. Qiao, G. Li, and X. Kong are with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, also with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China, and also with the Beijing National Research Center for Information Science and Technology, Beijing 100084, China (e-mail: zhangmh16@mails.tsinghua.edu.cn; junbi@tsinghua.edu.cn; qiaoy18@mails.tsinghua.edu.cn; ligy18@mails.tsinghua.edu.cn; kongxiao0532@gmail.com).

K. Gao is with the College of Cybersecurity, Sichuan University, Chengdu 610017, China (e-mail: godrickk@gmail.com).

Z. Li is with Baidu Inc., Beijing 100193, China (e-mail: lizhaogeng01@baidu.com).

H. Hu is with the School of Computing, Clemson University, Clemson, SC 29634 USA (e-mail: hongxih@clemson.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSAC.2019.2894189

[1]Apparently, NFV is not the only form for the software-based implementation, which achieve an on-demand provision and scaling using virtulization techniques. However, we believe that the software-based implementation could benefit from the NFV paradigm.

of traffic makes failure resilience even harder since volumes of the state or packets that need to be backuped are huge. Upon failures, the impacts on the normal packet processing are not negligible. A cloud gateway framework must provide ***resilient*** packet processing without introducing too much burden or performance penalty on regular packet processing.

While many existing studies [6]–[8], [10]–[16], [28], [29] have provided some insightful thoughts to build such a framework for cloud gateways, they only address part of these challenges and are not able to satisfy all three requirements simultaneously (details in Section II).

To overcome the limitations of existing studies, we propose TRIPOD, a novel network function framework, specialized for cloud gateways to fulfill the three requirements simultaneously. The core idea of TRIPOD is to jointly manage and optimize traffic, processing logic and state. In particular, TRIPOD uses *1) a simple, efficient traffic processing mechanism*, which is enabled by *a simple network configuration* and *a stateless, connection-oriented programming model*, to substantially obtain the traffic management scalability and enhance the packet processing performance (to satisfy requirement **R1** and **R2**), and *2)* HPSMS, a *high performance state management service*, which adopts multiple optimizations, such as *fast state lookup* and *predictive replica placement*, to leverage data locality and failure prediction and provide a traffic-aware state management service for near-optimal packet processing and efficient failure recovery (to satisfy requirement **R2** and **R3**).

We implement a prototype of TRIPOD and evaluated it using both real cloud gateway traffic traces and extreme pressure tests. The results demonstrate that TRIPOD can effectively achieve scalable traffic management, i.e., < 100 flow rules for a cluster with even $\sim$Tbps traffic, high performance, i.e., reducing 40% of latency compared with *StatelessNF* [12], and the similar failure resilience as *StatelessNF*, with reasonable overheads, i.e., less than 10% of the workload traffic.

Our **main contributions** in this paper are:

1) We identify the essential requirements for cloud gateways and systematically analyze why existing solutions fall short in handling cloud gateway traffic. (§II)

2) We propose TRIPOD, a novel, simple-to-use network function framework, specialized for cloud gateways to address identified challenges. The gain of TRIPOD comes from the joint management of traffic, processing logic and state, with a *simple, efficient traffic processing mechanism* and a *high performance state management service* (HPSMS). (§III)

3) We introduce two optimization techniques, namely *fast state lookup* and *min-churn predictive replica placement*, to substantially improve the performance and efficiency of HPSMS and achieve a traffic-aware state management service. (§IV, §V)

4) We implement a prototype of TRIPOD and evaluated it using both real cloud gateway traffic traces and extreme pressure tests. The results demonstrate that TRIPOD can effectively achieve scalable traffic management as well as near-optimal latency, throughput and failure resilience with reasonable overheads. (§VI)

In addition, we discuss some important issues in Section VII, summarize the related work in Section VIII and conclude our paper in Section IX.

II. MOTIVATION & OUR APPROACH

In this section, we discuss the limitations of existing NF frameworks and why they fall short at cloud gateways. Then we present our new approach, TRIPOD, and show the challenges as well as our observations to make it fit for the cloud gateway scenarios.

A. Limitations of Existing Work

We demonstrate the design space of how different frameworks make different decision choices in addressing the aforementioned challenges, and show the gap between the existing studies and the requirements of cloud gateways.

1) Unscalable Traffic Management: Taking the advantage of the fine-grained SDN traffic control, many studies [13], [15], [16], [29] choose to keep track of the location where the state is stored and direct the traffic to the corresponding location. While these frameworks have the potential to achieve high availability[2] with all state replication and backup, they do not take failure resilience into consideration, nor provide a concrete and optimized mechanism to handle failures. Moreover, given a huge amount of concurrent connections (up to millions [21]) and a very large address space (up to millions [23]) at cloud gateways, it is not practical to aggregate the flow rules on the switch, keep the information on a distributed SDN controller and steer the traffic timely and correctly. Because this is limited by the flow table size (up to thousands) and flow rule update rate (tens of milliseconds for a flow rule and no more than 200 updates per second) in state-of-the-art OpenFlow switches [30]. A scalable network traffic management approach is urgently needed to process the high-volume cloud gateway traffic.

2) High Performance Without Failure Resilience: Many network functions are implemented as virtual machines or native libraries on commodity servers. These applications or frameworks [6]–[8], [14] can achieve very efficient packet processing (i.e., $\sim$Mpps) with relatively simple network configurations. However, they do not take high availability into consideration and cannot extend to support failure resilience easily: state about connections is simply lost when a machine fails, and another server cannot serve flows migrated from another machine correctly. As machine failure happens frequently at cloud gateways [25], [26], the missing of fault tolerance guarantee obviously stops them from deployment. A high-performance solution with failure resilience is highly desired to fulfill the quality of service at cloud gateways.

3) Failure Resilience With High Costs: While approaches such as *Pico Replication* [10] and *FTMB* [11] can provide perfect fault tolerance for stateful network functions, they both come at the expense of high costs. *Pico Replication* introduces a substantial per-packet latency ($\sim$10ms), which obviously violates the latency requirement of cloud gateways [24].

[2]In this paper, we denote *failure resilience*, *availability* and *fault tolerance* as similar meaning.

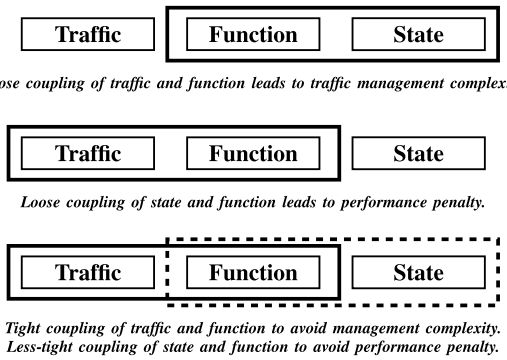


Fig. 1. Choices of Decoupling Traffic, Processing Logic and State.

FTMB requires packet duplication, which would occupy 50% of the total bandwidth if *Input Logger* is located at another server. It also incurs long recovery time due to packet replaying (up to hundreds of milliseconds). These two disadvantages make *FTMB* suffer from high overheads and substantial performance penalty upon failures. A recent trend to provide high availability for NFs is to decouple the packet processing logic from the network state [12], [18], [20], [31], and store the state in a remote, resilient data store. By trading perfect fault tolerance model for cost benefit (see §VII), these approaches could gain fast and efficient failure recovery. However, all these data stores are traffic unaware, which means the state is always steered to a remote, static place (data store) without regarding to the traffic distribution. Moreover, most of these systems (especilly StatelessNF [12]) usually require very high-end hardware such as InfiniBand [32] to reduce the performance penalty (e.g., per-packet latency due to remote access) as low as possible, making the deployment daunting for enterprises. Even so, the additional latency is as high as hundreds of microseconds [12], which is unacceptable at cloud gateways. Meanwhile, since each packet requires remote data access to read/write the network state, the packet number of control traffic is several times more than that of workload traffic, which makes the overhead considerable and the scalability of the remote data stores a significant concern. A low-cost failure resilience mechanism is in dire need for gaining high performance, low capital expense and high scalability at cloud gateways.

B. Our Approach: TRIPOD

The limitations of the aforementioned approaches motivate us to propose TRIPOD, a new framework customized for cloud gateways. We highlight two challenges that must be addressed carefully before TRIPOD could be fit for cloud gateways.

- **C1**: How to achieve high performance (large throughput and low latency) with correct traffic management in the extremely large traffic volume and address space?
- **C2**: How to accomplish efficient failure resilience with low cost and high scalability, *i.e.*, providing high availability without introducing too many overheads or performance penalty?

As demonstrated in Fig. 1, TRIPOD jointly manages *traffic*, *processing logic* and *state*. Especially, traffic and processing logic are tightly coupled that each packet is always processed on the server it arrives in, while processing logic and state are coupled in a less tight way: state might still need to be transmitted but the replicas are placed based on traffic prediction. As a result, we achieve the effect that *state is aware of where traffic will flow through under any circumstance*. In particular, we make the following specific design choices integrated with the unique characteristics of cloud gateways to address the challenges above.

To address the first challenge, TRIPOD proposes a *simple and efficient traffic processing mechanism*, which jointly optimizes the workload traffic and processing logic. Binding processing logic either to traffic or state requires to keep a mapping of where exactly the processing logic is placed, yet steering traffic is more complex and less efficient than migrating state because this is constrained by the hardware capabilities and traffic is more uncontrollable at cloud gateways. As a result, we resort to the simple network configuration, e.g., ECMP [33], which can achieve simple traffic management and be easily supported by almost all commodity switches in modern data centers. Besides, more resilient ECMP implementation could be adopted to avoid global traffic distribution to reduce failure penalty if advanced switches could be accessed. Moreover, we observe that servers at cloud gateways are usually under-utilized to reserve resources for traffic burst for high performance consideration, where the total capacity is at least twice the average workload in busy hours. As a result, we adopt a stateless, connection-oriented programming model, which takes advantage of *run-to-completion* (RTC) execution model and *traffic-processing logic co-location* to effectively enhance the packet processing performance.

To address the second challenge, TRIPOD proposes a *high performance state management service*, which aims at achieving efficient packet processing even when there are failures. Keeping state in a remote, resilient data store can achieve high availability, yet it incurs high latency and poor scalability. Thus, data locality is still important to achieve efficient packet processing and we want to leverage it as much as possible. An intuitive approach is to use the memory of each NF instance to act as the *cache* of the remote data store. However, it is not trivial to achieve this during failures, since traffic may experience *churns*,[3] which would lead to service degradation and slow failure recovery. To efficiently achieve data locality in cloud gateway packet processing, TRIPOD chooses to implement its own traffic-aware state management service instead of simply offloading state management to a remote, resilient data store. This state management service, namely HPSMS, is based on the idea of *predictive replica placement* that places state replicas to servers where traffic can potentially be directed when there are network failures. We also design several techniques to make the state service fast, robust and cost-efficient.

[3]Current ECMP implementation incurs unnecessary traffic redistribution during updates. When a next-hop is added or removed in ECMP, flows are reshuffled among NF instances, which is defined as *churns* in this paper. Even with advanced, resilient ECMP implementation, part of traffic still needs to be steered to (a) new NF instance(s).

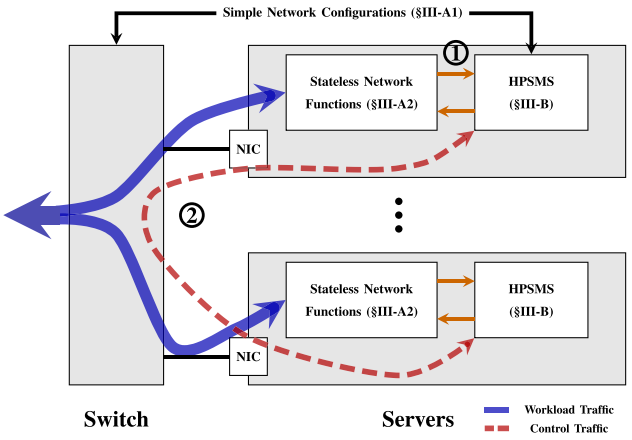


Fig. 2. TRIPOD architecture.

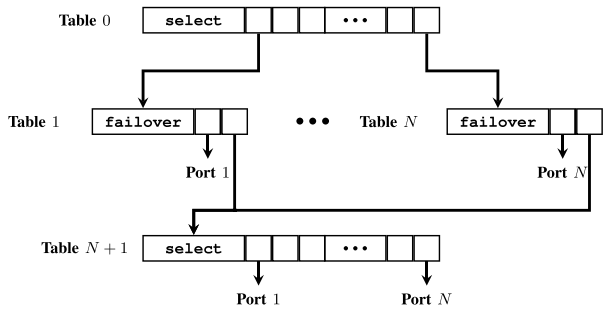


Fig. 3. Two-stage ECMP.

```
class NetworkFunction {
public:
    virtual void onInit(const Tuple &tuple,
                        Packet &pkt,
                        Direction direction) = 0;

    virtual void onPacket(const Tuple &tuple,
                          Packet &pkt,
                          Direction direction) = 0;

    virtual void onClose(const Tuple &tuple,
                         Packet &pkt,
                         Direction direction,
                         Reason reason) = 0;
};
```

```
class StateManagementService {
public:
    virtual void readTS(const Tuple &tuple,
                        int nf_id, size_t size,
                        void *data);

    virtual void writeTS(const Tuple &tuple,
                         int nf_id, size_t size,
                         void *data);
};
```

Fig. 4. API for TRIPOD network functions.

III. DESIGN OVERVIEW

As demonstrated in Fig. 2, we introduce the overall architecture of TRIPOD: the simple and efficient traffic processing mechanism (Section III-A), including the *simple network configuration* (Section III-A1), and the *stateless, connection-oriented programming model* (Section III-A2). And the high performance state management service (HPSMS) (Section III-B). The two optimizations of HPSMS, *fast state lookup* (①) and *min-churn predictive replica placement* (②) are discussed in Section IV and Section V, respectively.

A. Simple and Efficient Traffic Processing Mechanism

1) Simple Network Configuration: Each TRIPOD cluster is placed in a single rack where multiple NF nodes (servers) are connected to the ToR (Top-of-Rack) switch(es). In practice, there could be multiple links between an NF node and a switch, or an NF node could be connected to multiple ToR switches. However, they share much of the design as the simplest case where there is only one ToR switch and one link between the switch and each NF node.

The traffic is distributed to all servers in the rack using ECMP (Equal Cost Multi-Path) [33]. For switches supporting the *select* and *fast failover* group tables as defined in OpenFlow 1.3 [34] and above, we can optionally use two-stage ECMP, a variant of ECMP which helps to reduce churns. Two-stage ECMP is configured as demonstrated in Fig. 3. Each port k is allocated a `failover` group table (table k) with port k being the primary bucket and a secondary `select` group table (table $N+1$) as the failover bucket. The primary `select` group table (table 0) has N entries and each points to a unique `failover` group table. If a server is still connected to the switch, *i.e.*, the corresponding port is still up, two-stage ECMP will not redirect its traffic even if there are network failures, which avoids global traffic redistribution. If more advanced programmable switches could be obtained, e.g., Barefoot Tofino [35], Cavium XPliant [36], we could also adopt more resilient hashing algorithms to replace the simple $hash(5tuple)\%N$ in current ECMP implementation [33], such as consistent hashing [19] and stable hashing [20], which could load balance traffic well as well as ensure connection affinity when NF nodes fail.

With the switch configuration independent of the packet header space, and stateless network functions that are not bound to a specific machine (§III-A2), the management of network functions and traffic is both simple and scalable. It is also quite resilient to failures (*i.e.*, we do not get severe flow disruption or service degradation because of network reconfiguration or NF redistribution).

2) Stateless, Connection-Oriented Programming Model: The programming model of TRIPOD is motivated by other NFV frameworks, in particular StatelessNF [12] and Pico [10], where a network function itself does not keep persistent internal state but has access to two *logically separated* state tables[4]: 1) the *configuration table*; and 2) the *runtime state table*. The *configuration table* stores the configurations of both the NF runtime system, e.g., the service function chain for packets, and each NF instance, e.g., the access control list and candidate server list. It is relatively stable and is read-only to the NF. The *runtime state table* stores the runtime state of each connection.

Each NF is required to extend the base class as shown in Fig. 4 and to expose three function interfaces. The

[4]Although the state and processing logic are separated logically, they could be co-located physically to get high performance as in TRIPOD.

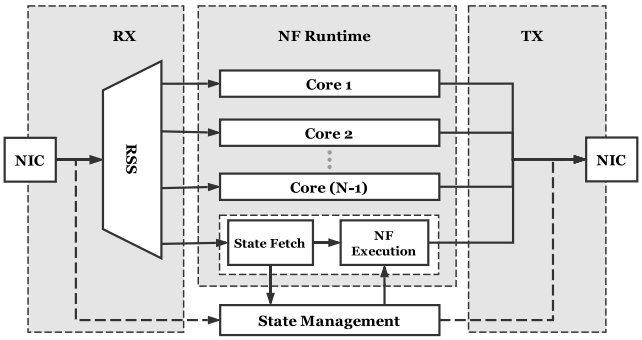


Fig. 5. NF execution model in TRIPOD.

`onInit` function and `onClose` function are invoked only once during the lifespan of a connection state – when a state is allocated[5] (`onInit`) and is about to be removed (`onClose`). The main logic should be implemented in `onPacket` function which is invoked whenever a packet with the same tuple arrives. The variables in these functions are basically some common information that can be extracted from the packets. The type `Tuple` contains the 5-tuple of $\langle src\_ip, dst\_ip, src\_port, dst\_port, proto \rangle$ and other information such as RSS (receiver-side scaling) hash results. To ensure that symmetric TCP connections always get the *same* RSS results on all NF nodes, we use the key given by Woo and Park [37]. The type `Packet` wraps the `sk_buffer` and provides methods to access the fields quickly. `Direction` indicates whether the packet is an incoming packet (from the Internet to the data center) or an outgoing one (from the data center to the Internet). The `Reason` type is specific to the `onClose` function which indicates why the state is removed, including `TCP_CLOSED` (the TCP connection is closed), `TIMEOUT` (inactive for a long time) and `TERMINATED` (explicitly removed by an NF).

To get a higher performance, TRIPOD adopts an *all-in-one* design for network function management and deploys all network functions on all servers, similar to *NetBricks* [8] and *BESS* [38]. In other words, a packet is processed in a *run-to-completion* (RTC) manner, *i.e.*, it goes through all NF instances in the service function chain in a single core. Since resources at cloud gateways are abundant, the scaling of the entire service function chain can be easy and efficient. This execution model is demonstrated in Fig. 5. The incoming traffic from an NIC is distributed to multiple RX queues using DPDK's RSS configured with the aforementioned key. Each queue is only accessed by a single core so that packets from a single TCP connection are always processed on the same core.

B. High Performance State Management Service

In this section, we introduce the HPSMS subsystem and describe how our HPSMS subsystem achieves high performance, high availability and low overheads.

[5]We allocate state differently for TCP and UDP. For TCP connections, the state is allocated only when a SYN packet is received. For UDP, we allocate the state whenever the tuple is never seen before from both directions.

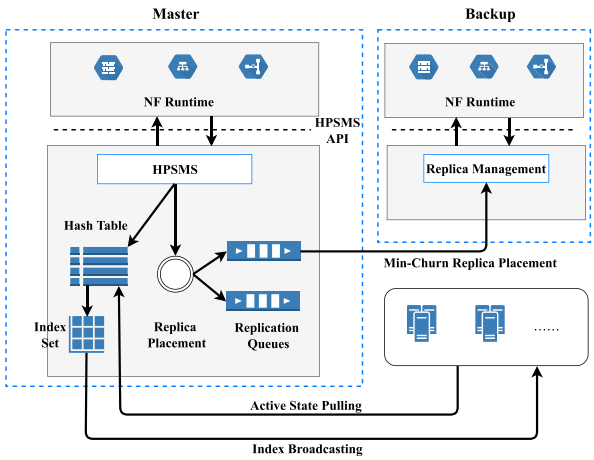


Fig. 6. HPSMS architecture.

TABLE I
MESSAGES IN HPSMS

Name	*Description*
Confirm(msgs)	Confirm messages *msgs* (including responses)
State Lookup	
PushIndex(s, I)	Push (broadcast) the index set I from s to all other servers (§IV)
PullIndex(d)	Pull incremental index sets from all to d (§IV)
Replication	
PushReplica(s, d, L)	Push a list of state replicas L from s to d
PullReplica(s, d, I)	Pull replicas for index set I from s to d
Failover Recovery	
Predict(c, e)	Predict the potential next hop for connection c during failure e (§V)

As shown in Fig. 6, just like general fault-tolerant distributed storage systems, HPSMS also has the concept of *master* node and *backup* node. The roles of nodes in HPSMS are based on which connection is of concern, *i.e.*, a node can *be a master node for certain connections and a backup node for other connections at the same time*. Connection state is stored in an in-memory hash table and is replicated from a master node to one or more backup nodes using logs. Some basic control messages are shown in TABLE I. It is noteworthy that these messages or interfaces are transparent to the NF programmer, in other words, they are used to achieve the HPSMP mechanism under the hood. We have designed a special DPDK driver so that the control traffic (logs transmission) can share the same NIC with the NF runtime. This in-band control design allows us to work on machines that only have one NIC, which simplifies the network setup and allows us to do certain optimizations.

We enforce the data locality that guarantees that each packet is processed on the master node of its connection, *i.e.*, the HPSMS subsystem guarantees that there exists a copy of the state in the local state table. While data stores such as RAMCloud [39] achieve data locality by always routing the client operations for a given key to the same *master* node,

our system works in a reversed way: *whenever a node sees a packet, it becomes the master node of the connection which the packet belongs to*. The packets of the same connection will always be sent to the same node under our resilient network configuration, unless there are failures. When failures happen, traffic might redistribute and packets can arrive on another node, which is known as churns. Upon churns, the nodes in HPSMS start switching their roles. We further adopt the following techniques to enhance the performance and reduce the overhead of HPSMS.

1) Fast State Lookup: Fast state lookup answers the question of *how state should be accessed by network functions*, which play a critical role in the overall performance. Careless design of HPSMS could incur performance penalty and high overheads, even suffering from connection disruption and potential attacks. We conduct two optimizations, *lookup aggregation* and *index broadcasting*, to enhance the performance, robustness and reduce the overheads. The details are discussed in Section IV.

2) Min-Churn Predictive Replica Placement: Min-churn predictive replica placement answers the question of *where the replica should be placed*. This is an important approach to reduce the churns caused by ECMP traffic redistribution. The underlying idea is that *if a packet arrives on a backup node, the node can become a master node without the delay/traffic of replica transmission*. Min-churn predictive replica placement is the main factor that HPSMS achieves fast recovery, and we discuss the details in Section V.

3) Optimized Replica Transmission: Optimized replica transmission answers the question of *how the replica can be transmitted efficiently*. Two techniques are used in HPSMS: batching and piggy-backing.

With min-churn predictive replica placement, we have multiple replication queues where the logs in the same queue have the same destination. Merging multiple logs in a single packet can improve the goodput and reduce network traffic.[6]

Another technique is piggy-backing. Since the replication can happen between all node pairs, the confirmation packet of replica packet from node i to node j carries a batch of logs from node j to node i.

IV. Fast State Lookup

The time to conduct a state lookup operation is directly related to latency and throughput. Thus, efficient state lookup is crucial to the overall performance of softwarized cloud gateways. In this section, we demonstrate the techniques used in HPSMS to achieve this goal.

A. Lookup Aggregation

Lookup aggregation aims at reducing the number of lookups to improve the packet processing performance. In our execution model, packets from the same flow will be processed by a fixed service chain where each network function needs to perform a lookup operation. Conducting the lookup operation independently can lead to a large latency and redundant control traffic, especially during network failures because the state has to be fetched from a remote server.

[6]Batching could incur additional latency and compromise the consistency of fault tolerance model. For scenarios that require stricter consistency model, batch size could be set as 1.

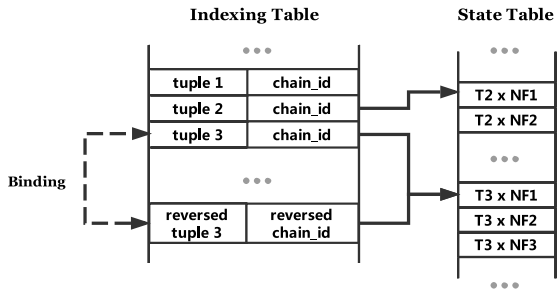


Fig. 7. Tuple-based memory layout.

Based on the observation that *the state used by all NFs in the same chain is associated with the same flow*, HPSMS introduces a tuple-based memory layout (Fig. 7) to eliminate the redundant lookup operations. The state are stored in a two-layered structure. The states associated with the same flow are placed together in the state table, and the address is stored in the indexing table with a `chain_id` which indicates the version of service function chain when it is first accessed. TRIPOD fetches all the state before the packets are sent to the subsequent network functions.

Cloud gateway network functions (e.g., network address translation) may rewrite the packet headers. As a result, the inbound traffic and the outbound traffic may not go to the same server. For an incoming packet, HPSMS keeps the binding between the 5-tuple as it enters the network function chain and the 5-tuple as it leaves *the last network function* (which is the first network function that handles outbound traffic). The latter is called a *reversed tuple*. It points to the same local state and is either normally replicated to the correct server or pulled by the server beforehand during the remote lookup.

B. Remote Lookup With Index Broadcasting

Remote lookup operations require communications among servers and are expensive. Unnecessary remote lookup operations might potentially lead to service degradation such as longer latency and lower throughput, and even be exploited by attackers to paralyze the system.

Given the network configuration in our system, there are three scenarios where an NF server v can encounter a missing entry for a connection c (identified by its 5-tuple): *1)* the packets belong to the outbound traffic; *2)* there is a network failure e and there is no replica for c on v; and *3)* an attacker creates packets for a connection that does not exist.

In the first case and the second case where e is recoverable, there will be at least one valid state on a remote server. An intuitive approach to obtain the state is *broadcasting*, that is, the NF node broadcasts a state-request packet to all the other nodes to pull the state. However, this would incur heavy load to the control traffic, and may be exploited

Algorithm 1: Fast State Lookup

```
Input: i - the core ID, c - connection, direction - in/out
1  Function LOOKUP(i, c, direction)
2      if (c, direction) ∈ local_hash_map then
           // Local hit
3          APPEND(working_queue[i], (c, direction))
4      else if (c, direction) ∈ local_key_set then
           // Replica is on a remote machine
5          candidates ← INDEXSETLOOKUP((c, direction))
6          ASYNCFETCH(candidates, (c, direction))
7      else
           // Either attacking traffic or out
              of sync, buffered for batch
              request
8          APPEND(request_queue[i], (c, direction))
9          if |request_queue[i]| ≥ threshold then
10             ASYNCREQUESTINCREINDEXSET
               (i, request_queue[i])
11             request_queue[i] ← ∅
```

by the attacks to trigger the amplification attack[7] to paralyze the HPSMS subsystem.

To address this, we propose the idea of *index broadcasting*. The basic idea is to let an HPSMS node have the global view and enable it to conduct local *lookup* on entries in a remote server. This is achieved in TRIPOD by broadcasting the indices (*i.e.*, the set of keys) on a single node actively to all other nodes. Besides eliminating potential attacks, *index broadcasting* also enables a global synchronization among different nodes, which could guarantee the correctness even when the prediction makes mistake. As the number of connections is pretty large, broadcasting the indices as they are is not feasible and we *compress* the indices using Cuckoo filter [40]. The index sets of incoming traffic and outgoing traffic are broadcasted separately.

Since the broadcasted index sets are not always up to date, there is still a race condition that the packet just arrives too soon. Making complete queries may be exploited by attackers, making the system vulnerable. Instead, we batch these packets and send queries for *incremental index broadcasting*. If the packet is from a valid connection, its state must be included in the incremental index sets so we can safely discard the rest.

The final state lookup is described in Algorithm 1. It first checks if there is a local copy (Lines 2-3), and then leverages the index set to do fast remote lookup (Lines 4-6). If both lookup operations fail, it requests for incremental index sets in batches (Lines 7-11).

V. MIN-CHURN PREDICTIVE REPLICA PLACEMENT

Even though the system can already work very efficiently with the previous components, we want to further reduce the system overheads, *e.g.*, number of replicas in the system and traffic for replica migration. In this section, we introduce the *min-churn predictive replica placement*, an important feature of HPSMS to achieve these goals.

A. Perfect Predictive Replica Placement

The key observation is that *if the new server already has a replica of a connection's state during network failures, no state migration is required*. Thus, for a given connection c and a given failure scenario e (a list of failed servers/links), if we know which server it will be forwarded to and make that server a backup node beforehand, the packets of the connection can be processed smoothly when the failure scenario really happens.

We denote this server as $b(c, e)$ where $b : C \times E \mapsto S$ is referred to as a *prediction function* which maps a connection and an error to a specific server. We also denote the set of actual backup nodes for connection c as $B(c)$ which requires $|B(c)|$ replicas.

For a *perfect predictive replica placement* or *zero-churn predictive replica placement* which requires zero state migration for failures up to k servers, we have:

Proposition 1: For a given connection c in an N-node HPSMS cluster and a given prediction function b, the number of replicas to achieve perfect (zero-churn) predictive replica placement for failures up to $k(k < N)$ server(s) is $|B_0(c)|$, where

$$B_0(c) = \left\{ b(c, e) \;\middle|\; e \in \bigcup_{i=1}^{k} C(N, i) \right\},$$

and $C(N, i)$ represents the i-combinations of N.

One can easily conclude that $|B_0(c)| \geq k + 1$ because we need at least $k + 1$ replicas to survive a k-failure. In the worst case, $|B_0(c)| \sim O(k^2)$, because for all redirected flows to find a local replica of its state, we need at most $\sum_{i=1}^{k}(i + 1) = \frac{k(k+2)}{2}$ replicas. To find a balance between churns during failure recovery and the normal replication traffic, we introduce *min-churn predictive replica placement*.

B. Min-Churn Predictive Replica Placement Algorithm

The key idea of *min-churn predictive replica placement* is to select $k+1$ replicas such that *1)* it can survive all failures up to k servers, and *2)* it minimizes the worst-case churn traffic during these failures.

For each connection c, let the size of its connection state be s_c. We use a binary variable r_c^j to indicate that whether c has a state replica on server j. We measure the worst-case churn as the maximum expectation of traffic volume received on a single server for replica migration, which can be represented as:

$$\varphi = \max_j \left\{ \sum_{e}^{1 \leq |e| \leq k} P(e) \left(\sum_{c}^{b(c,e)=j} s_c - \sum_{c}^{b(c,e)=j} r_c^j s_c \right) \right\}, \quad (1)$$

where $P(e)$ represents the probability that failure e happens.

Thus, we have the objective of *min-churn predictive replica placement* being $\min \varphi$,

[7]One missing entry could lead to several control messages, so we call it amplification attack.

subject to:

$$\sum_j r_c^j = k+1, \quad \forall c, \tag{2}$$

$$r_c^{b(c,\emptyset)} = 1, \quad r_c^j \in \{0,1\}, \ \forall j, \ \forall c, \tag{3}$$

where Equation 2 represents the constraint that we have at most $k+1$ replicas and Equation 3 represents that there always exists a replica on the master node (the original copy).

However, how this optimization can be conducted in practice is not trivial. First, the number of c is very large (the number of all concurrent connections). Second, it is not preferred to migrate replicas which are already placed on some backup node to others. Thus, we introduce an algorithm to practically compute the replication plans.

We first rearrange the objective function in Equation 1:

$$\begin{aligned}\varphi &= \max_j \left\{ \sum_e^{1\le|e|\le k} P(e) \left(\sum_c^{b(c,e)=j} s_c - \sum_c^{b(c,e)=j} r_c^j s_c \right) \right\} \\ &= \max_j \left\{ \sum_i \left(\sum_c^{b(c,\emptyset)=i} \left(\sum_e^{1\le|e|\le k} P(e) b_{cj}^e s_c \left(1 - r_c^j\right) \right) \right) \right\} \\ &= \max_j \left\{ \sum_i \sum_c^{b(c,\emptyset)=i} b_c^j s_c (1 - r_c^j) \right\}, \end{aligned} \tag{4}$$

where b_{cj}^e is an indicator for $b(c,e) = j$ and $b_c^j = \sum_e^{1\le|e|\le k} P(e) b_{cj}^e$.

An additional constraint is added that connections whose r_c^j is already known will not change the replica placement. For these connections (say $C_i^{(l)}$), $\sum_i \sum_{c\in C_i^{(l)}}^{b(c,\emptyset)=i} b_c^j s_c (1 - r_c^j))$ is known and we denote it as $S_j^{(l)}$. Equation 4 becomes

$$\varphi^{(l+1)} = \max_j \left\{ S_j^{(l)} + \sum_i \sum_{c\notin C_i^{(l)}}^{b(c,\emptyset)=i} b_c^j s_c (1 - r_c^j) \right\}. \tag{5}$$

Let $x_c^j = 1 - r_c^j$, minimizing Equation 5 is equivalent to $\min \rho$,

subject to:

$$\begin{aligned} & S_j^{(l)} + \sum_i \sum_{c\notin C_i^{(l)}}^{b(c,\emptyset)=i} b_c^j s_c x_c^j \le \rho, \quad \forall j \\ & \sum_j x_c^j = N - k - 1, \quad \forall c \\ & x_c^{b(c,\emptyset)} = 0, \quad x_c^j \in \{0,1\}, \ \forall j, \ \forall c. \end{aligned} \tag{6}$$

We first relax the constraint that x_c^j must be binary so that this relaxed min-max optimization problem can be solved using the distributed algorithm introduced by Notarnicola *et al.* [41], and then use rounding methods to get the actual values of all $\{x_c^j\}$. Note that this procedure can be quickly finished within millisecond on a normal server cluster. Since we don't have all the x_c^j because c keeps coming in, we take a batch of B connections and conduct this optimization as demonstrated in Algorithm 2.

Algorithm 2: Min-Churn Predictive Placement

```
1  Function onConnection(i, c, s_c)
2      if queue.size ≤ B then
           // Batch B connections
3          queue.add(c)
4          record s_c
5          b_c^j ← AsyncPredict(c)
6      else
           // Replicate to remote servers
7          ρ_i, {r_c^j} ←
           SolveLP(queue, {s_c}, {λ_ij}, {μ_j}, {S_j})
8          foreach r_c^j ≠ 0 do
9              Replicate(i, j, c)
10         queue ← ∅
11         ExchangeParameters({λ_ij}, μ_i, S_i)
12 Function onParameterExchange({λ_ij}, {μ_j}, {S_j})
13     record {λ_ij}, {μ_j}, {S_j}
```

C. Predicting ECMP Next-Hops

We now introduce how next hops (*i.e.*, $b(c,e)$) for ECMP or two-stage ECMP can be predicted. The basic idea is to *simulate* the traffic redistribution using the switch with *forged* failures and *forged* packets.

1) To predict failures with up to K failures, configure the switch to use K extra ECMP groups, numbered as $g_1, \ldots, g_K$. These groups are matched on VLAN id so that the original data center traffic would not use any of these groups. For the g_k, it contains exactly $N-k$ ports.
2) When a server determines that the state of a connection c should be replicated, it "forges" K packets using the same 5-tuple but different VLAN numbers and sends them to the network with the *Predicate* message. Here each e is actually encoded as an integer $k = |e|$, the number of failed servers.
3) If a server receives such a packet, it extracts e from the message and looks up its ID in ECMP group g_e which is statically stored on all servers. It sends a *Confirm* message to the *Predicate* message with the ID.
4) After the sender receives confirmation for all K failures, it takes a k and the corresponding id (denoted as b_k). Now the server can simulate all k-failures and $b(c,e)$ is the b_k-th server in the remaining working machines in the primary ECMP group.

The prediction process is demonstrated in Fig. 8. When failures happen, the prediction may not be accurate anymore. If it is confirmed that a failure may not recover very soon, we update the ECMP groups used by prediction and the corresponding mappings on servers. The network configuration for workload traffic is not affected.

VI. Evaluation

In this section, we seek to answer the following key questions with extensive evaluations:

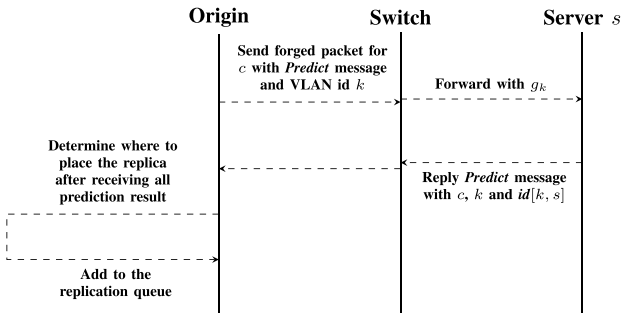


Fig. 8. ECMP prediction.

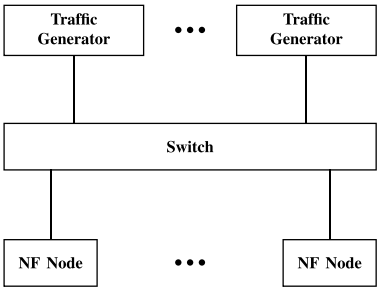


Fig. 9. Physical topology for the experiment.

- How scalable is TRIPOD in terms of traffic management? (§VI-B)
- How does TRIPOD perform overall in terms of latency, throughput, and failure resilience? (§VI-C)
- How does a single machine in TRIPOD perform? (§VI-D)
- How useful are the techniques and optimization we adopted in TRIPOD? (§VI-E)
- What are the overheads of TRIPOD? (§VI-F)

A. Methodology

1) Topology: Our experiments are conducted using the network topology shown in Fig. 9. T traffic generators and K identical TRIPOD servers are connected to a single ToR switch (Pica8 P-3922, 48×10 GbE $+ 4 \times 40$ GbE). Unless otherwise specified, all servers are equipped with Intel(R) Xeon(R) E5-2600 v4 CPUs (2.4 GHz, 2 NUMA, each with 6 physical cores and 12 logic cores), 15360K L3 cache, 64G RAM and an Intel 82599ES 10GbE NIC. All servers are installed with Ubuntu 14.04 LTS with Linux kernel 4.4.0-31-generic.

2) Traffic Generation: We use a real-world inter-DC traffic trace collected from a cloud gateway server of Baidu Inc., one of the large cloud provider around the world. Real IP addresses are obscured but the trace still retains the relationship between different connections. The trace is replayed with *Pktgen* [42] to simulate the real-world scenario of a cloud gateway. Since the trace itself is collected on a single machine, we amplify it by making K copies and change the IP addresses with different prefixes to simulate the traffic for the entire cloud gateway.

3) Metric Collection: We collect the metrics every $100ms$.

- **Latency**: Latency is measured on traffic generators which actually represents the round-trip time of a packet entering and leaving the TRIPOD cluster.
- **Throughput**: Throughput is measured as the number/size of the packets processed on each server and the total throughput is the sum of throughput on all servers.
- **Broken connection**: Broken connections are measured as those whose packets are dropped because of missing state. These connections are already established on end hosts but the packets can never go through the cloud gateway correctly because the state is lost.
- **Overhead**: Overheads are measured by the number/volume of control messages, including state replication, ECMP prediction and index broadcasting.

4) Implementation: Each NF node in TRIPOD, including NF runtime, HPSMS and several typical network functions at cloud gateways (TABLE II), is implemented in DPDK 17.08.1 with $\sim$5k lines of code.[8] All of the three parts above are located at the same NUMA for high performance. We compare our prototype system with the state-of-the-art practice, where state is separated from processing logic and stored in a remote, reliable storage, including StatelessNF [12], Beamer [20], Yoda [18] and Protego [31]. While the latter three only focus on one simple network function, load balancer, StatelessNF [12] can support more network functions, which is indeed more general. Since the source code of StatelessNF [12] is not available, we have adapted our prototype system to keep the state in a remote, resilient data store, just as the original paper. We refer to this implementation as *StatelessNF* to understand the benefit of state-computation co-location. Further, we also implement a system without any fault tolerance guarantees as a performance baseline (referred to as *Baseline*).

B. Scalable Traffic Management

We first demonstrate the scalability of TRIPOD's network management by the number of flow rules. We have configured the switch with two-stage ECMP for 48 ports and ECMP prediction for up to 10 failures. As expected, it only consumes 60 flow rules. On the other hand, state-of-the-art SDN-based load balancing solutions such as [43] already take up hundreds to thousands for a university-scale network, which potentially leads to poor scalability at cloud gateway scenarios.

We also demonstrate how it can support dynamic horizontal scaling. Horizontal scaling means the ability to add/remove servers. Since removing a server is equivalent to a failure case which we discuss in later sections, we only show the case where a server is dynamically added. We first configure the traffic generator to generate traffic of 22 Gbps with 4 TRIPOD servers. Other two servers are connected to the switch but initially DPDK does not handle any packet so the port is not detected as active. After 4 and 12 seconds respectively, we launch a new instance of TRIPOD on each of the two servers. The throughput of each machine during the process[9] is demonstrated in Fig. 10. As we can see, loads on the four

[8]We do not use the virtulization techniques in all our experiments for the high performance consideration at cloud gateways. Nevertheless, packaging our network functions in containers or virtual machines is also feasible.

[9]The redistribution can cause a small burst of control messages, but the effect is similar to a failure and is analyzed later.

TABLE II
NETWORK FUNCTIONS AT CLOUD GATEWAYS

Network Functions	*State*	*Access Patterns*
Network Address Translator [16]	Address pool, NAT entry	1 Write Per-connection, 1 Read Per-packet
Load Balancing [21]	Address pool, LB entry	1 Read/Write Per-connection
Stateful Firewall [12]	Static configurations, Per-connection state	5 Read/Write Per-connection, 1 Read Per-packet
Monitor [8]	Static configurations, Packet counter	1 Read/Write Per-packet

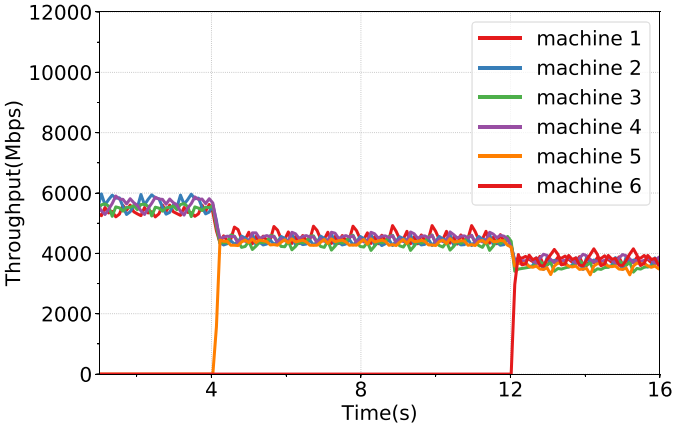


Fig. 10. Per-machine throughput change during horizontal scaling (adding two new servers).

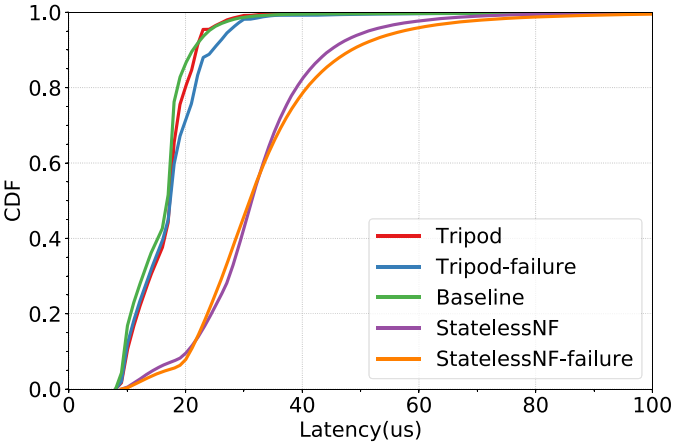


Fig. 11. Latency before/after a single failure.

existing machines first drop to about 4/5 after $4s$ and then 2/3 after $12s$, while the throughput of new instances quickly grows up to almost the same level, showing that **TRIPOD can support dynamic horizontal scaling in a very instantaneous and seamless way**.

C. Overall Performance

We now demonstrate the overall performance of TRIPOD including *latency*, *throughput*, and *failure resilience*. All evaluations are conducted on a 4-machine cluster and we shut down the DPDK application on one machine to simulate a single-node network failure.

1) Latency: Fig. 11 demonstrates the cumulative distribution function (CDF) of latency for TRIPOD, the baseline and StatelessNF. As we can see, the latency of TRIPOD is very close to the performance baseline, while StatelessNF results in larger latency (almost *twice the latency* of TRIPOD and the baseline). This is reasonable because StatelessNF needs to do a remote lookup for each packet, which requires another round trip from one server to another. The results demonstrate that TRIPOD benefits substantially from state-computation co-location.

2) Throughput: We do not use an extremely large traffic volume here due to the limited capability of our traffic generation tools. Note that this is also not necessary since the overall throughput of TRIPOD could be estimated as the number of TRIPOD machines times the throughput of each single machine. From the experiment in the next subsection (§VI-D), we can see that a single server in TRIPOD could

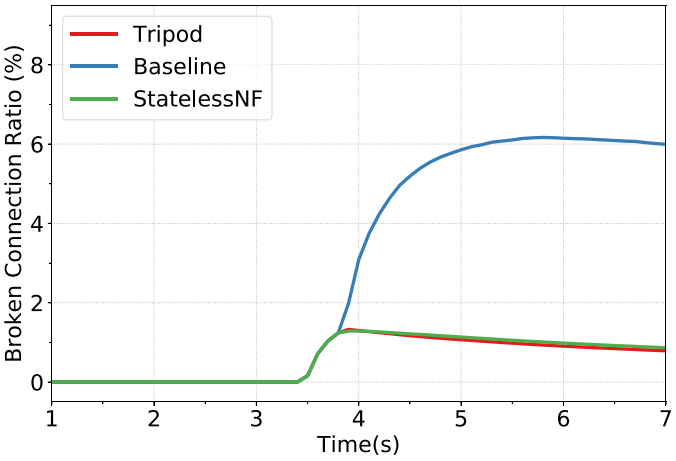


Fig. 12. Broken connection ratio.

prcocess almost 40Gbps workload traffic, and a TRIPOD cluster is able to reach as high as ∼Tbps level with only 25 servers. In this subsection, we mainly focus on how the throughput changes when failures happen. We can see that three frameworks have similar throughput during normal traffic, both in total (Fig.13(a)) and on each machine (Fig.13(b), 13(c) and 13(d)). We can also see that after failure, TRIPOD and StatelessNF both achieve about 4/3 of the original throughput, while the throughput of the baseline drops because of ECMP churns. In our evaluation, there is a gap between the time when the failure actually happens and when the other servers start to react. This is because the Pica8 switch uses a

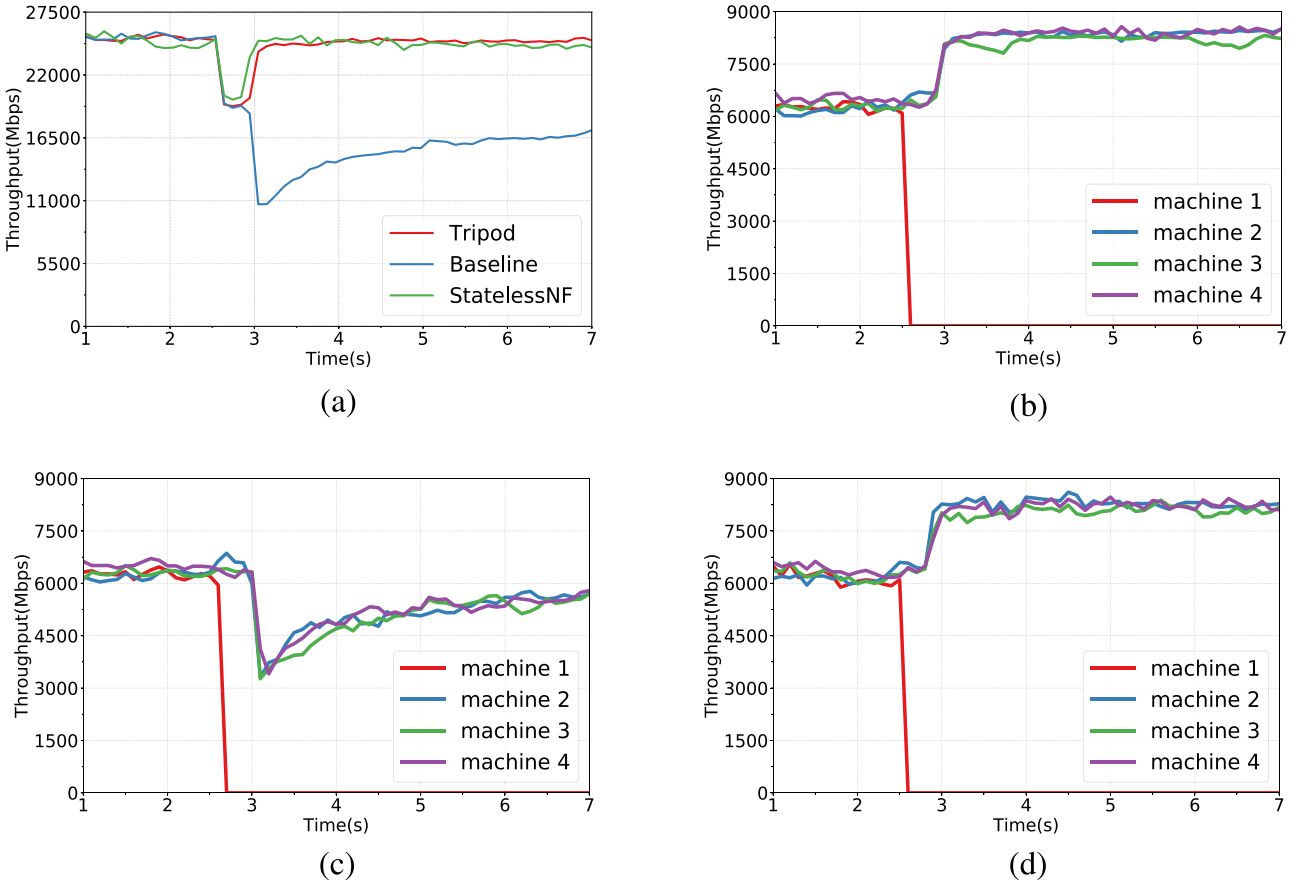


Fig. 13. Total and single-machine throughput for different solutions in a single failure. (a) Total throughput. (b) TRIPOD. (c) Baseline. (d) StatelessNF.

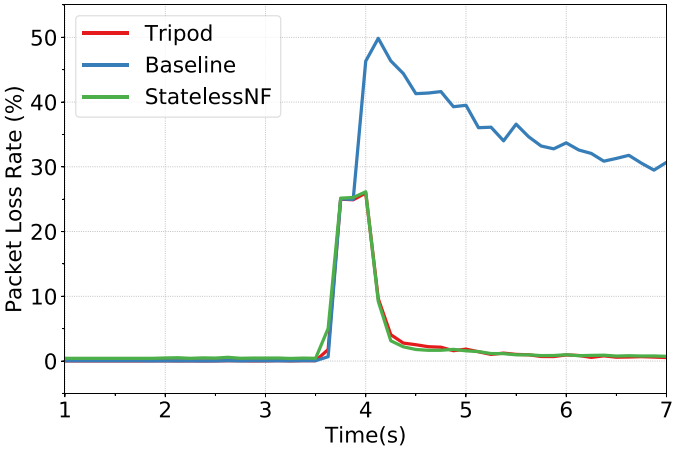


Fig. 14. Packet loss rate.

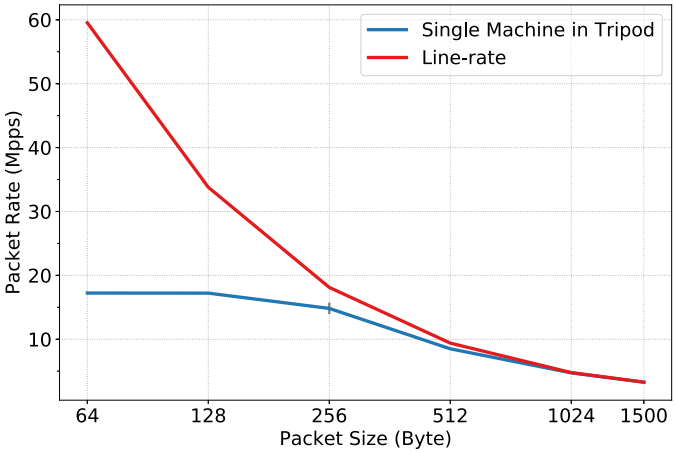


Fig. 15. Single NF Node Throughput vs. Packet Size.

built-in 250ms for port liveness[10] and it is possible to reduce this gap by setting a smaller detection time.

3) Failure Resilience: We now demonstrate the failure resilience results. As shown in Fig. 12, since TRIPOD and StatelessNF adopt the same fault tolerance model, the proportion of broken connections of TRIPOD and StatelessNF is similar, which is only 1/4 of the baseline. If a link/node failure happens every day, cumulative results indicate that TRIPOD can provide an availability of 99.9999%. We also analyze the impacts on packet loss, since it might potentially affect the performance of TCP connections. As shown in Fig. 14, the baseline has two large leaps while TRIPOD and StatelessNF only have one. The first one is because of the network failure so that all traffic to the failed node is lost (roughly 1/4). The second leap of baseline is because of ECMP churns which no longer exist in stateless solutions. We can see that TRIPOD almost overlaps with StatelessNF, indicating that TRIPOD achieves similar failure resilience with state of the art.

[10]We have contacted Pica8 technical support and the 250ms is confirmed by their engineers. Right now the value is not configurable in PicaOS and manually disabling the port does not eliminate the gap. However, this feature might become available in the future.

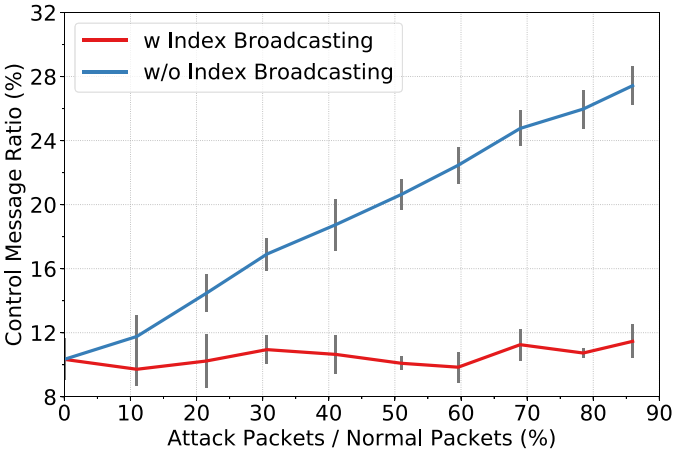


Fig. 16. Control Traffic/Workload Traffic w/o index broadcasting.

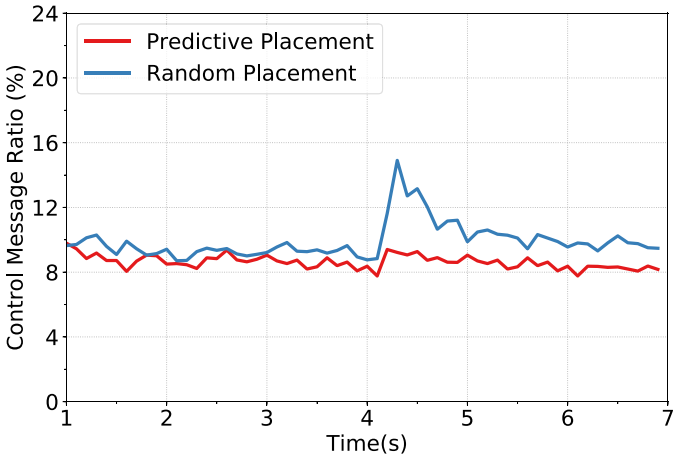


Fig. 17. Control Traffic/Workload traffic.

D. Single Machine Performance

Since each server in TRIPOD receives a roughly equal amount of traffic through ECMP, the overall throughput of TRIPOD could be estimated as the number of TRIPOD machines times the throughput of each single machine. The more traffic each server can handle, the fewer machines will be required to serve the same amount of traffic capacity.

Since a single machine could easily reach the upper limit of the 10 GbE NIC capability (14.88Mpps for the minimum packet), we change the origin 10 GbE NIC to an Intel XL710 40GbE NIC, and use the 40 GbE port of Pica8 P-3922. We use different size of packets to benchmark the throughput of a single machine in TRIPOD. As shown in Fig. 15, TRIPOD is able to forward the minimum packets at 17.3Mpps, and the rate of larger packets is restricted by the physical NIC capability (40Gbps). The overall throughput for minimum packets in original StatelessNF [12] paper is 4.6Mpps, which is only 1/4 of a single machine in TRIPOD, demonstrating that state-computation co-location and lookup aggregation enhance the performance of the NF nodes in TRIPOD greatly. A more prominent benefit comes from the distribution property of our HPSMS subsystem. In the original StatelessNF [12], the data store, RAMCloud, has a coordinated server which serves all the requests and distributes them to the other servers, and could easily be a bottleneck. In contrast, we do not have such an explicit coordinated server in HPSMS, and each server serves its own requests, which can achieve extremely high scalability. Note that the average packet size in the data center is about 850 bytes [44], [45], the throughput for this packet size can easily reach the theoretical limit of 40 GbE NIC.

E. Micro Benchmarks

We now conduct various experiments to demonstrate the effectiveness of our techniques and optimizations.

1) Index Broadcasting: Index broadcasting can enable a global view synchronization for each NF node and guarantee the correctness even if unexpected events happen, especially, it can effectively enhance the robustness of HPSMS. As shown in Fig. 16, with index broadcasting, the number of control messages remains unchanged when the rate of attack packets increases, while without index broadcasting, each attack packet results in a remote broadcasting and the control messages increase linearly. The result demonstrates index broadcasting can effectively mitigate the amplification attack.

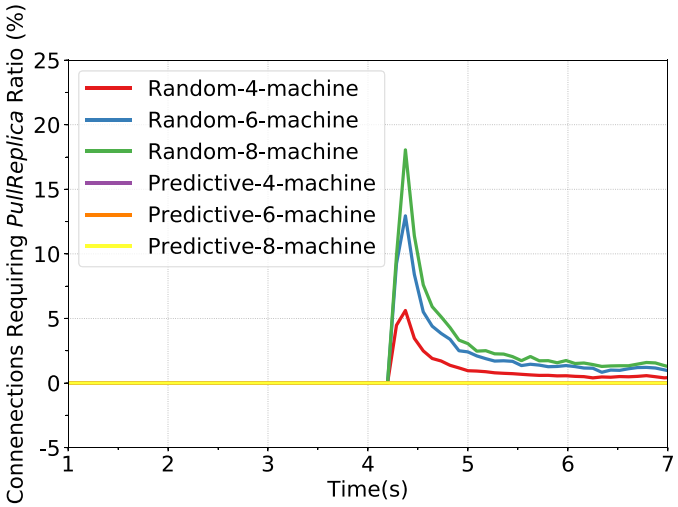


Fig. 18. Connections requiring *PullReplica*.

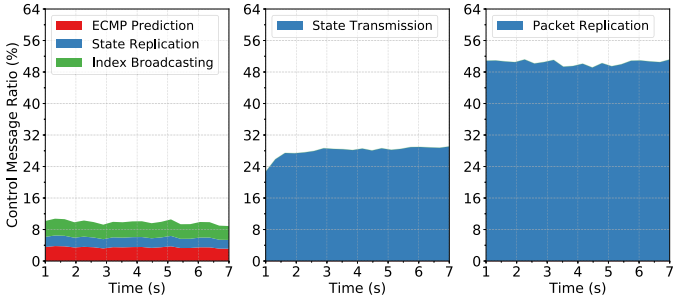


Fig. 19. Control Traffic/Workload traffic of TRIPOD, *StatelessNF* and *FTMB*.

2) Min-Churn Predictive Replica Placement: To demonstrate the effectiveness of min-churn predictive replica placement, we compare it with a variant of TRIPOD using random replica placement. As we can see in Fig. 17, the randomized variant incurs an increase of control message when a failure happens, while the number of control messages remains

relatively unchanged for the predictive placement. Our initial experimental setup is with 4 NF instances, and the difference of TRIPOD and the randomized variant is not so obvious because flows still have a high probability of arriving on a backup machine. We increase the number of machines and count the proportion of connections which require *PullReplica*. As we can see in Fig. 18, the proportion of random replica placement grows significantly, while TRIPOD, using predictive replica placement, eliminates the overheads. Given the trend, we expect predictive replica placement to be much more efficient when deployed on tens of servers.

From the discussion above, we can see that **the optimizations can effectively enhance the performance, robustness and reduce the overheads of TRIPOD** .

F. Overheads

We now demonstrate the overheads of TRIPOD, *i.e.*, bandwidth required for state replication, ECMP prediction and index broadcasting. As we see from Fig. 19, the ratio of control traffic consumes about 8-10% of the workload traffic. This is a huge improvement compared with per-packet fault tolerance approaches such as FTMB [11], which require half of the total bandwidth for packet duplication when *Input Logger* is located at another server. Meanwhile, StatelessNF introduces about 28% traffic overheads, almost 3 times larger than TRIPOD. Thus, we conclude that **the overheads of TRIPOD is reduced greatly compared with state-of-the-art approaches**.

VII. DISCUSSION

A. Per-Connection Model

Current TRIPOD mainly supports and optimizes stateful network functions with per-flow state. On the one hand, the per-flow model is sufficient to support almost all in-line mode network functions at cloud gateways. Traffic at cloud gateways is extremely large, and it must be separated to multiple NF instances to achieve distributed processing (usually in 5-tuple flow grain). As a result, each NF instance could only see part of the traffic, which inherently results in the per-flow state view of network functions. Although there may be some sophisticated network functions with cross-flow state (e.g., IDS, DPI, DoS detection) deployed at cloud gateways, they do not work in an in-line mode [46], and they do not have such a rigid latency requirement and such a large portion as in-line mode-based network functions, making it essential to do some specific optimizations and achieve both high performance and high availability for network functions with per-flow state. On the other hand, TRIPOD could also support cross-flow model. Per-connection state could be processed in the paradigm as it is in TRIPOD, and cross-flow state could be transmitted to a coordinated server to achieve a cross-machine synchronization, just like StatelessNF [12]. The discrimination of cross-flow and per-flow states could be conducted with methods such as StateAlyzr [29]. Another approach is to jointly optimize traffic management and processing logic. With more advanced network configurations than TRIPOD, packets of the same cross-flow state are distributed as a whole to the same NF instance. As a result, cross-flow state could be managed in a similar way as HPSMS. We leave the exploration of more general HPSMS system with optimization for cross-flow state to our future work.

B. Fault Tolerance Model

We illustrate this concept from the following two aspects. First, we only assume the NF nodes would fail unexpectedly in TRIPOD. On one hand, server failures are a common assumption which has been confirmed by researchers through measurement, while the ToR switch is highly reliable [25], [26]. On the other hand, even if a switch fails unexpectedly, it does not cause any connection disruption. This is because multiple NF nodes (servers) are usually connected to multiple ToR (Top-of-Rack) switches in a rack, and the ToR switches are stateless and interchangeable. Second, TRIPOD shares the same fault tolerance model as StatelessNF [12]. This fault tolerance model can not recover from the situation when the packets have been transmitted to the *fail-stop* instance, which means these in-transmit packets may lead to a slight inconsistency between the original instances and the backup instances. However, as we discussed with the cloud provider, this fault tolerance model could make a balance between the cost and benefit, and it fits for the cloud gateway scenarios. Although *Pico Replication* [10] and *FTMB* [11] could provide perfect fault tolerance model, their expenses such as per-packet delay and per-packet duplication either violate latency requirement, or incur high overheads and substantial performance penalty, which is unacceptable for cloud gateways.

VIII. RELATED WORK

Beyond the most directly related work discussed in Section II, we discuss other related work based on the following four categories.

A. Elasticity of NFV Instance

Split/Merge [13], OpenNF [15] and S6 [16] focus on transferring state between NF instances to achieve correct and efficient NF elasticity. These frameworks cannot support failure resilience well while TRIPOD puts failure resilience as the top priority. U-Haul [28] reduces state migrations by only migrating the elephant flows while StateAlyzr [29] uses standard program analysis tools to automatically identify the state objects that need to be managed during redistribution. Both U-Haul and StateAlyzr are orthogonal with our work and can potentially improve the performance of TRIPOD.

B. NF Instances at Cloud Scale

Several systems have been proposed for cloud-scale softwarized network functions. Ananta [17], Yoda [18], Maglev [19] and Beamer [20] provide softwarized load balancing. Ananta [17] employs ECMP to build a scale-out system and uses a highly available control plane to coordinate state across the distributed data plane. Yoda [18] uses memcached, an in-memory key-value database to store the state of TCP connections. Maglev [19] uses consistent hashing (with a unique hash function) and connection tracking (*i.e.*,

the bindings between VIP and the server for active TCP connections) to achieve highly reliable packet delivery even under failures. Beamer [20] leverages the per-connection state already held by back-end servers to guarantee occasional stray connections are never dropped under churns. Protego [31] provides IPSec tunnels at the cloud scale by carefully separating the control plane state (keys) and data plane state of IPSec and storing them in different locations. It also leverages the renegotiation of keys to achieve dynamic VM migration and efficient IPSec consolidation. While these systems are highly specialized for a certain application with high performance and high availability, TRIPOD can support more applications, making it useful as a general framework for cloud gateways. Note that Ananta [17], Maglev [19] and Beamer [20] also adopt ECMP for traffic distribution, and the role MUX plays is somewhat similar to that of NF nodes in our paper. However, there are several distinguished differences between TRIPOD and these load balancing systems. First, they all assume difference MUXes are interchangeable, that is, once a MUX fails, other MUXes could seamlessly reconstruct the VIP-DIP mapping state by themselves. This is because all MUXes adopt the same DIP selection algorithm, and the selection is only decided by the 5 tuples of the packets. In contrast, network functions at cloud gateway such as stateful firewall do not follow this practice, the state of these network functions are updated by nearly all packets of a flow. As a result, for a persistent flow, if the state triggered by previous packets are lost, the subsequent packets of the flow could not be processed correctly, which means these dynamic state must be carefully managed to ensure the packets of the same flow are processed equally even when NF node failures occur. Second, Ananta [17] and Maglev [19] are sensitive to the changes of back-end servers, and Beamer [20] eliminates this concern with a stateless MUX design and daisy chaining. TRIPOD handles this concern with an efficient and resilient state management service (HPSMS) for NFs, and the TRIPOD system is transparent to both ends of a connection. When a back-end machine fails, only connections pertained to it will be affected. When adding a new back-end server, only new connections will be directed to this new server to gradually achieve a new balance. Besides, our efficient and resilient state management service are much more general, which can leverage more sophisticated network functions, not only simple load balancing as stated in these papers.

C. Failure Recovery for In-Memory Storage Systems

RAMCloud [39] uses log-structured replicas which stores replicas on the disks in multiple backup servers. Cocytus [47] uses a hybrid recovery mechanism which uses primary backup for metadata and keys, and uses erasure coding for values. TRIPOD has adopted many useful design features from these studies. However, instead of providing a general-purpose highly available storage system, our HPSMS subsystem is deeply integrated into the overall TRIPOD framework and has leveraged the characteristics in traffic management. This joint design makes it possible to achieve better performance and also reduce overheads.

D. SDN/NFV at 5G Scenario

Ksentini *et al.* [48] use Nash Bargaining game and the threat point to seek for the optimal SDN controller placement points in 5G. Dutra *et al.* [49] adopt SDN to enable operators to efficiently allocate the network resource with multi-paths routing, which further guarantees the required end-to-end QoS. Taleb *et al.* [50] introduce several different VNF placement algorithms to achieve diverse optimization goals for virtual 5G network infrastructure. Bagaa *et al.* [51] propose three heuristic algorithms to find the optimal network function placement locations in carrier cloud. Addad *et al*. [52] benchmark the ONOS intnt interfaces to ease 5G service management. Addad et al. [53] formulate a MILP optimization model to enable a cost-optimal network slice deployment for 5G. Bagaa et al. [54] devise an efficient VNF placement algorithm to sustain the QoS and reduce the deployment cost. These works target at different domain and adopt different techniques from ours.

IX. CONCLUSION

In this paper, we introduce TRIPOD, a novel, simple-to-use framework, which is specially designed for cloud gateways. To provide *high-performance* and *failure-resilience* packet processing with a *scalable traffic management* mechanism, TRIPOD jointly manages *traffic*, *processing logic* and *state*. In particular, TRIPOD uses a *simple, efficient traffic processing mechanism* and a *high performance state management service* (HPSMS), with two critical techniques, namely *fast state lookup* and *min-churn predictive replica placement*, to substantially improve the performance, scalability and reduce the overheads. Our evaluation results demonstrate that TRIPOD achieves seamless horizontal scaling, low latency, large throughput, and high availability with reasonable overheads, making it a good fit for cloud gateways.

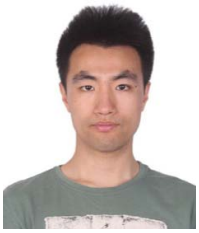
REFERENCES

[1] Microsoft. (2017). *Microsoft Azure Cloud Computing Platform & Services.* [Online]. Available: https://azure.microsoft.com/en-us/
[2] Rackspace. (2017). *Rackspace: Managed Dedicated & Cloud Computing Services*. [Online]. Available: https://www.rackspace.com
[3] Amazon. (2017). *Amazon Web Services (AWS)—Cloud Computing Services*. [Online]. Available: https://aws.amazon.com
[4] Alibaba. (2017). *An Integrated Suite of Cloud Products, Services and Solutions Alibaba Cloud.* [Online]. Available: https://www.alibabacloud.com/
[5] Google. (2017). *Google Cloud Computing, Hosting Services & APIs*. [Online]. Available: https://cloud.google.com/
[6] J. Martins *et al.*, "Clickos and the art of network function virtualization," in *Proc. NSDI*, 2014, pp. 459–473.
[7] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: High performance and flexible networking using virtualization on commodity platforms," *IEEE Trans. Netw. Service Manage.*, vol. 12, no. 1, pp. 34–47, Mar. 2015.
[8] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV," in *Proc. OSDI*, 2016, pp. 203–216.
[9] Y. Hu, M. Song, and T. Li, "Towards full containerization in containerized network function virtualization," *ACM SIGOPS Oper. Syst. Rev.*, vol. 51, no. 2, pp. 467–481, 2017.
[10] S. Rajagopalan, D. Williams, and H. Jamjoom, "Pico replication: A high availability framework for middleboxes," in *Proc. SoCC*, 2013, p. 1.
[11] J. Sherry *et al.*, "Rollback-recovery for middleboxes," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 227–240, 2015.

[12] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *Proc. NSDI*, 2017, pp. 97–112.
[13] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes," in *Proc. NSDI*, vol. 13, 2013, pp. 227–240.
[14] S. Palkar *et al.*, "E2: A framework for NFV applications," in *Proc. SOSP*, 2015, pp. 121–136.
[15] A. Gember-Jacobson *et. al.*, "OpenNF: Enabling innovation in network function control," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 163–174, 2014.
[16] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic scaling of stateful network functions," in *Proc. NSDI*, 2018, pp. 213–299.
[17] P. Patel *et al.*, "Ananta: Cloud scale load balancing," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 207–218, 2013.
[18] R. Gandhi, Y. C. Hu, and M. Zhang, "Yoda: A highly available layer-7 load balancer," in *Proc. EuroSys*, 2016, p. 21.
[19] D. E. Eisenbud *et al.*, "Maglev: A fast and reliable software network load balancer," in *Proc. NSDI*, 2016, pp. 523–535.
[20] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless datacenter load-balancing with beamer," in *Proc. NSDI*, vol. 18, 2018, pp. 125–139.
[21] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proc. SIGCOMM*, 2017, pp. 15–28.
[22] R. Gandhi *et al.*, "Duet: Cloud scale load balancing with hardware and software," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 27–38, 2015.
[23] AzureSpeed. (2018). *Datacenter IP Range*. [Online]. Available: http://www.azurespeed.com/Information/IpRange
[24] Juniper. (2018). *SRX5400, SRX5600, and SRX5800 Services Gateways*. [Online]. Available: https://www.juniper.net/assets/us/en/local/pdf/datasheets/1000254-en.pdf
[25] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 350–361, Aug. 2011.
[26] K. V. Vishwanath and N. Nagappan, "Characterizing cloud computing hardware reliability," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 193–204.
[27] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.
[28] L. Liu, H. Xu, Z. Niu, P. Wang, and D. Han, "U-haul: Efficient state migration in NFV," in *Proc. 7th ACM SIGOPS Asia–Pacific Workshop Syst.*, 2016, p. 2.
[29] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella, "Paving the way for NFV: Simplifying middlebox modifications using statealyzr," in *Proc. NSDI*, 2016, pp. 239–253.
[30] X. Jin *et al.*, "Dynamic scheduling of network updates," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 539–550, 2014.
[31] K. Tan, P. Wang, Z. Gan, and S. Moon, "Protego: Cloud-scale multitenant ipsec gateway," in *Proc. ATC*, 2017, pp. 473–485.
[32] IBTA. (2017). *InfiniBand Trade Association: Home*. [Online]. Available: http://www.infinibandta.org/ and http://www.infinibandta.org/
[33] C. E. Hopps. *RFC 2992: Analysis of an Equal-Cost Multi-Path Algorithm*. Assedded: 2018. [Online]. Available: https://tools.ietf.org/html/rfc2992
[34] O. N. Fundation, "Openflow switch specification version 1.3.3," Tech. Rep., 2013.
[35] Barefoot Networkss. (2017). *Barefoot: World's Fastest P4-Programmable Ethernet Switch Asics*. Accessed: Jul. 13, 2018. [Online]. Available: https://barefootnetworks.com/products/brief-tofino/
[36] XPliant. (2018). *Xpliant Ethernet Switch Product Family*. Accessed: Jul. 19, 2018. [Online]. Available: https://www.cavium.com/xpliant-ethernet-switch-product-family.html
[37] S. Woo and K. Park, "Scalable TCP session monitoring with symmetric receive-side scaling," KAIST, Daejeon, South Korea, Tech. Rep., 2012.
[38] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "SoftNIC: A software NIC to augment hardware," Tech. Rep. UCB/EECS-2015-155, 2015.
[39] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast crash recovery in ramcloud," in *Proc. SOSP*, 2011, pp. 29–41.
[40] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proc. CoNext*, 2014, pp. 75–88.
[41] I. Notarnicola, M. Franceschelli, and G. Notarstefano. (2016). "A duality-based approach for distributed min-max optimization." [Online]. Available: https://arxiv.org/abs/1611.09168
[42] AppNeta. (2017). *PKTgeen—PCAP Editing and Replaying Utilities*. [Online]. Available: http://tcpreplay.appneta.com/
[43] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford, "Efficient traffic splitting on commodity switches," in *Proc. CoNext*, 2015, p. 6.
[44] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," in *Proc. 1st ACM Wsorkshop Res. Enterprise Netw.*, 2009, pp. 65–72.
[45] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. IMC*, 2010, pp. 267–280.
[46] ipwithease. (2018). *Firewall vs IPS vs IDS*. [Online]. Available: https://ipwithease.com/firewall-vs-ips-vs-ids/
[47] H. Chen *et al.*, "Efficient and available in-memory KV-store with hybrid erasure coding and replication," *ACM Trans. Storage*, vol. 13, no. 3, p. 25, 2017.
[48] A. Ksentini, M. Bagaa, and T. Taleb, "On using SDN in 5G: The controller placement problem," in *Proc. GLOBECOM*, 2016, pp. 1–6.
[49] D. L. C. Dutra, M. Bagaa, T. Taleb, and K. Samdanis, "Ensuring end-to-end QoS based on multi-paths routing using SDN technology," in *Proc. GLOBECOM*, 2017, pp. 1–6.
[50] T. Taleb, M. Bagaa, and A. Ksentini, "User mobility-aware virtual network function placement for virtual 5G network infrastructure," in *Proc. ICC*, 2015, pp. 3879–3884.
[51] M. Bagaa, T. Taleb, and A. Ksentini, "Service-aware network function placement for efficient traffic handling in carrier cloud," in *Proc. WCNC*, 2014, pp. 2402–2407.
[52] R. A. Addad, D. L. C. Dutra, M. Bagaa, T. Taleb, H. Flinck, and M. Namane, "Benchmarking the ONOS intent interfaces to ease 5G service management," in *Proc. GLOBECOM*, 2018.
[53] R. A. Addad, T. Taleb, M. Bagaa, D. L. C. Dutra, and H. Flinck, "Towards modeling cross-domain network slices for 5G," in *Proc. GLOBECOM*, 2018.
[54] M. Bagaa, T. Taleb, A. Laghrissi, and A. Ksentini, "Efficient virtual evolved packet core deployment across multiple cloud domains," in *Proc. WCNC*, 2018, pp. 1–6.

Menghao Zhang received the B.S. degree from the Department of Computer Science, Tsinghua University, China, where he is currently pursuing the Ph.D. degree with the Institute for Network Science and Cyberspace. His research interests include software-defined networking, network function virtualization, and cyber security.

Jun Bi (S'98–A'99–M'00–SM'14) received the B.S., C.S., and Ph.D. degrees from the Department of Computer Science, Tsinghua University, Beijing, China. He is currently a Changjiang Scholar Distinguished Professor of Tsinghua University and also the Director of the Network Architecture Research Division, Institute for Network Sciences and Cyberspace, Tsinghua University. He successfully led tens of research projects, published more than 200 research papers and 20 Internet RFCs or drafts, and owned 30 innovation patents. His current research interests include Internet architecture, SDN/NFV, and network security. He is a Distinguished Member of China Computer Federation.



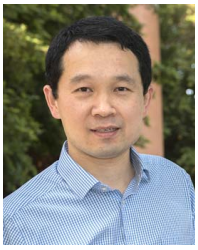
Kai Gao received the B.S. and Ph.D. degrees from the Department of Computer Science and Technology, Tsinghua University. His research interests include software-defined networking and distributed systems.

Yi Qiao received the B.S. degree from the Department of Computer Science, Tsinghua University, China, where he is currently pursuing the master's degree with the Institute for Network Science and Cyberspace. His research interests include software-defined networking, network function virtualization, and cyber security.

Guanyu Li received the B.S. degree from the School of Computer Science and Technology, Huazhong University of Science and Technology, China. He is currently pursuing the Ph.D. degree with the Institute for Network Science and Cyberspace, Tsinghua University. His research focuses on software-defined networking, network function virtualization, and cyber security.

Xiao Kong is currently pursing the master's degree with the Institute of Network Science and Cyberspace, Tsinghua University, and the bachelor's degree in computer science with Nankai University. His research focuses on software-defined networking, network function virtualization, and cyber security.

Zhaogeng Li received the B.S. and Ph.D. degrees from Tsinghua University. He is currently a Senior Engineer with Baidu Inc. His main research interest includes datacenter network, RDMA, information-centric network, and edge computing.



Hongxin Hu (S'10–M'12) received the Ph.D. degree in computer science from Arizona State University, Tempe, AZ, USA, in 2012. He is currently an Assistant Professor with the Division of Computer Science, School of Computing, Clemson University. He has published more than 80-refereed technical papers, many of which appeared in top conferences and journals. His current research interests include security in emerging networking technologies, security in Internet of Things, security and privacy in social networks, and security in cloud and mobile computing. He was a recipient of the Best Paper Awards from ACM CODASPY 2014 and ACM SIGCSE 2018, and the Best Paper Award Honorable Mentions from IEEE ICNP 2015, ACM SACMAT 2016, and ACM SACMAT 2011. His research has also been featured by the IEEE Special Technical Community on Social Networking and received wide press coverage, including ACM TechNews, InformationWeek, Slashdot, and NetworkWorld.