NetStar: A Framework for Building Asynchronous Middleboxes

Paper #154, 13 pages

Abstract

1. Introduction

The trend of Network Function Virtualization (NFV) [2] aims to replace hardware middleboxes with software middleboxes running in virtualized environment. NFV greatly facilitates the deployment and provisioning of key network services. Since software middleboxes must be placed on critical network paths, such as backbone of LTE network [6], the performance of software middleboxes must be good enough to process packets at 10/40Gbps line rate.

Besides being high performance, the ability to handle asynchronous operations is also very important to NFV. First, some middlboxes need to collaborate with each other by passing requests and responses. For instance, the middleboxes on the control plane of IMS system [1] exchange a large number of SIP [3] requests and responses to establish a IP voice call. Second, middleboxes sometimes need to contact external service while processing flows. Stateless network function [4] stores critical flow states on an external key-value store [5], for scalability and resilience. Middlebox that handle application-level protocols, like the middleboxes on the control plane of IMS system [], need to query a DNS server to identify the next-hop middlebox instance to contact with. To achieve good performance, all these middleboxes that are mentioned are preferably to handle asynchronous operations in a fully non-blocking manner.

However, when implementing middleboxes with the state-of-art technique, non-blocking asynchronous operations can not be gracefully handled in a way that is both efficient and manageable. Today, most high-performance middleboxes bypass the kernel networking stack and process packets completely in user space with user space packet I/O frameworks, such as DPDK [] and Netmap []. The user space packet I/O frameworks assign several worker threads to busily poll the network interface card (NIC) for packets, so that the overhead caused by context switches when calling traditional kernel I/O system calls is completely avoided. The use of a busy poll loop makes user-spacke packet I/O framework both efficient and easy to program with. But when incorporating asynchronous operations into the user-space packet I/O framework, a dilemma is encountered. If one would like the middlebox implementation to be easily manageable, he can replace asynchronous operations with synchronous blocking operations, but that will seriously damange the runtime performance of middleboxes, as high-performance middleboxes can not even tolerate the overhead brought by context switches. If one would like the middlebox implementation to be highly efficient, he can achieve the goal with a combination of mutable state and callback functions, but this makes middlebox implementation hard to manage, as a middlebox implementation may require to query a key-value store several times in order to process a single packet [].

There are several existing frameworks that aim to provide asynchronous event processing for middleboxes, including mOS [], libnids [] and etc. These frameworks expose TCP related flow events to the middlebox programmer and enable these events to be processed in a non-blocking asynchronous manner. However, these frameworks are still based on callback-based design, making the code harder to manage when handling complex middlebox logic. They only concentrate on handling flow-level events that are related to TCP/IP protocol and they are not general enough to handle other asynchronous operations like database query.

In this paper, we present NetStar, a framework for implementing middleboxes that perform non-blocking asynchronous operations. NetStar is built upon open-source asynchronous library Seastar [] and provides basic constructs for building middleboxes. When compared with previous frameworks, NetStar exhibit the following advantages.

NetStar handles asynchronous operations of middleboxes in a way that is both efficient and manageable. Asynchronous operations in NetStar are accomplished through through callbacks, making NetStar highly efficient. However, the callbacks in NetStar are used in an implicit way that mimics the style of synchronous operations, making them easy to program with and reason about. NetStar's power comes from the promise-continuation programming

model provided by Seastar [] and advanced C++ features, such as lambda expression [].

NetStar has good multi-core scalability and performance. NetStar achieves multi-core scalability with hardware assistance. The input network traffic are automatically distributed to each core by NIC using RSS [], so that so that different cores can operate in a fully parallel fashion without contending for shared resources. NetStar is programmed with C++14 using various zero-cost abstractions. Even though the promise-continuation model that NetStar uses to implement asynchronous operations do pose slight runtime overhead, NetStar is able to compensate the overhead with multi-core scalability and achieve line-rate processing.

Using NetStar, it would be easy to implement traditional asynchronous operations such as DNS querying. Instead, we use NetStar to solve some important research problems that are raised recently.

First, we use NetStar to implement a non-blocking version of stateless network function []. Stateless network function [] has demonstrated great potential for dynamic scaling and resilience. However, its current implementation is blocking and synchronous, making its maximum performance inferior when compared to other state-of-art middlebox systems []. We use NetStar to transform stateless network function into a non-blocking and asynchronous one and pairs it with a key-value store [] with better performance than RamCloud []. Besides significantly improving the performance of stateless network function, we further identify a race problem when accessing the shared state stored on a key-value store by stateless network function. Without using NetStar, it would be hard to solve such a problem.

Second, we use NetStar to serialize access to shared data structures of middleboxes. Traditional middleboxes need to use lock to serialize the access to a shared data structures from multiple worker threads. We treat shared data structures as a dedicated service and convert the regular access-after-lock pattern to send-query-wait-response pattern. The new pattern is reasonably fast when compared to the access-after-lock pattern. It provides two additional benefits. First, it helps us solve the shared state access problem of stateless network function mentioned above. Second, it enables the middlebox to record the access order of any shared variables, so that another middlebox can replay the recorded access order in a deterministic fashion. We use this feature to build a primary-backup replication system for middleboxes. It's primary and back instances are always in synchronization and it has zero recovery time after primary failure.

Due to the use of new programming model and massive use of C++14 feature, porting existing middleboxes to NetStar may require non-trivial amount of work. However, as a new framework for exploring how asynchronous operations can be implemented in middleboxes, porting existing middleboxes is not our primary goal. We also believe that the constructs provided by NetStar for building middleboxes are general enough to implement other middleboxes that do not perform asynchronous operations.

The paper is organized as follows. We introduce our primary motivation in section 2. The overall architecture of NetStar is given in section 3. We discuss the design of asynchronous stateless network function and serialized shared data structure access in section 4. The performance of asynchronous stateless network function and a NAT is discussed in section 5. We discuss related work in section 6 and conclude the paper in section 7.

2. Motivations

Our primary motivation for this paper comes from a recent advancement in building stateless network function []. Stateless network function [] achieves dynamic scaling and fault tolerance, two of the most important topics that are active explored by network middlebox research community, by storing important flow states, including per-flow state or shared state, in a key-value store called RamCloud []. In the most extreme cases, stateless network function needs to access the key-value store for every other packet that it processes [].

Even though StatelessNF is probably the middlebox that has the most strigent requirement for asynchronous data store accessing, StatlelessNF is currently implemented in a blocking synchronized fashion: whenever the worker thread needs to access the data store, the worker thread has to be blocked and wait for the access to finish. A sample code of a stateless IPS is shown in figure **??**.

```
1  void stateless_ips(packet* pkt){
2    flow_tuple tp = extract_flow_tuple(pkt);
3
4    if(pkt_is_tcp_syn(pkt)){
5      automaton_state a_st = init_automaton_state();
6      sync_write_db(tp, a_st);
7    }
8    else{
9      automaton_state a_st = sync_read_db(tp);
10     char* next_byte = get_next_payload_byte(pkt);
11
12     while(next_byte != nullptr){
13       a_st = process_with_automaton(next_byte, a_st);
14     }
15
16     sync_write_db(tp, a_st);
17     send_pkt(pkt);
18   }
19 }
```

(a) Blocking synchronous IPS implementation from StatelessNF.

```
1  void on_first_pkt(context* ctx, packet* pkt){
2    flow_tuple tp = extract_flow_tuple(pkt);
3    automaton_state a_st = init_automaton_state();
4
5    async_write_db_with_cb(ctx, tp, a_st,
         subsequent_pkt_read);
6  }
7
8  void subsequent_pkt_read(context* ctx, packet* pkt){
9    flow_tuple tp = extract_flow_tuple(pkt);
10   async_read_db_with_cb(ctx, tp, subsequent_pkt_write
         );
11 }
12
13 void subsequent_pkt_write(context* ctx, packet* pkt,
       automaton_state* a_st){
14   char* next_byte = get_next_payload_byte(pkt);
15
16   while(next_byte != nullptr){
17     a_st = process_with_automaton(next_byte, a_st);
18   }
19
20   async_write_db_with_cb(ctx, tp, a_st, call_send_pkt
         );
21 }
22
23 void call_send_pkt(packet* pkt, context* ctx){
24   send_pkt(pkt);
25   update_ctx(ctx, subsequent_pkt_read);
26 }
```

(b) A callback based implementation for stateless IPS.

Figure 1: The API and implementation of an example NF module.

References

[1] 3GPP specification: 23.228. `http://www.3gpp.org/ftp/Specs/html-info/23228.htm`.

[2] Network functions virtualisation. `http://www.3gpp.org/DynaReport/23228.htm`.

[3] SIP: Session Initiation Protocol. `https://www.ietf.org/rfc/rfc3261.txt`.

[4] M. Kablan, A. Alsudais, E. Keller, and F. Le. Stateless network functions: Breaking the tight coupling of state and processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 97–112, Boston, MA, 2017. USENIX Association. ISBN 978-1-931971-37-9. URL `https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/kablan`.

[5] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, et al. The ramcloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 33(3):7, 2015.

[6] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker. Verifying reachability in networks with mutable datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 699–718, Boston, MA, 2017. USENIX Association. ISBN 978-1-931971-37-9. URL `https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/panda-mutable-datapaths`.