NetStar: A Framework for Building Asynchronous Middleboxes

Paper #154, 13 pages

Abstract

1. Introduction

Network Functions (NFs) are more than simple packet processors that perform stateless header transformation for each received packet, modern NFs need to carry out stateful processing for L4 TCP/UDP flows. To implement efficient L4 flow processing, NF must fully utilize flow-level asynchrony: after processing a packet for a single flow, the NF can move on to process packets for other flows without worrying about disturbing the original flow. Various NF software [2, 4, 5] uses event-driven programming to address flow-level asynchrony. After finish processing a flow, a micro context is saved for the flow and the NF switches to process other flows. When interested events happen for the flow, i.e. new TCP payload arrives, the saved context is resumed and a pre-determined callback functions are called to serve the event.

Besides flow-level asynchrony, modern NFs even need to address request-level asynchrony by contacting external services, so that they can be resilient to failures [6], collaborate with other NFs [1] and retrieve encrypted information [8]. For instance, StatelessNF [6] needs to retrieve important flow session state from a remote database [7] when processing the packets, so that even if the NF instances are failed, they can be quickly recovered. To ensure high-performance, the NF should not halt block waiting for the response, it should switch to process other flow packets, or handle other requests. Event-driven programming has a long history for addressing request-level asynchrony: the application registers a callback function to a pollable interface (Linux epoll). When the response arrives, the pollable interface invokes the callback function to handle the response.

The callback-based event-driven programming model has a long successful history, however, it still has some major draw backs when being used to develop NFs. First, the packet access pattern of a NF sometimes requires multiple asynchronous operations to be chained together in order to process a single packet [6]. This requires defining multiple callback functions and saving multiple contexts, which may significantly increase the number of the lines of code used for implementing simple NF logic. Second, due to the use of multiple callback functions, the control flow of the program is disrupted, making it hard to write and reason about the correctness of the resulting NF logic. Third, exception handling in existing event-driven framework can be repetitive, as each callback function needs to carefully handle possible error conditions. Finally, most existing event-driven programming framework is based on C programming language, which does not expose a safe programming interface. When the callback function is invoked, the callback function can literarlly modify arbitrary program state, which may crash the entire NF program if not carefully programmed.

In this paper, we proposed NetStar framework, which is designed to improve asynchronous programming in NF software. In particular, 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 [3] and advanced C++14 features, such as lambda expression and move semantics.

Based on the advanced programming model, we design a general purpose asynchronous flow abstraction that can precisely capture a wide rage of real-world NF requirements. Using the asynchronous flow abstraction, the processing task of the flow can pause at any time to perform asynchronous operations and resume normal processing when the asynchronous operation finishes. Our model is memory safe and only exposes events that the user registers.

We build several NFs with NetStar that capture a wide range of research and real-world needs, including an improved StatelessNF re-implementation, an HTTP proxy, an IDS and a SIP proxy used in IMS system. We show that NFs implemented in NetStar delivers sufficiently good performance in terms of packet processing throughput and la-

tency. Moreoever, we quantitatively evaluate the quality of the source code of NFs implemented using NetStar with NFs implemented using traditional call-backed event-driven programming model. Our quantitative study shows that for NFs that need to perform asynchronous operations, NetStar can significantly reduce the lines of code needed to implement the NF, reduce the number of the function definitions and reduce the lines of code that are devoted to error handling.

We make the following contribution in this paper:

First, we extend Seastar into a new framework for efficiently building asynchronous NFs. Second, we define a general purpose asynchronous flow model that can capture the requirement of a wide-range of real-world middleboxes. Finally, we use NetStar to build some practical middleboxes and quantitatively evaluate its effectiveness in reducing the lines of implementation code.

2. Motivations

In this section, we discuss the major motivations for creating NetStar.

2.1 Callback vs Promise

There has been a long lasting discussion about whether programmer should write callback functions to deal with asynchronous operations or use advanced programming techniques like promises []. The dominant conclusion is that using promise can significantly reduce application development effort by reducing the source code line count and maintaining a neat control flow.

Besides simple stateless packet transformers such as IP TLL adjuster and L2 switch, there is complicated NF software that needs to constantly perform asynchronous operations. Therefore, we believe that bringing promise into NFV development to address both flow and request-level asynchrony is important for NFV development as well.

Even though promises bring apparent advantages over callback functions, people rarely use promises to develop NF software. The primary reason is due to the complexity for supporting promises. Promise has a monadic structure [] and requires that the implementing language has a strong type system to support generic type arguments, type deduction and first-class lambda function. For a long time, promises are only seen in functional programming languages like OCaml [], F-sharp [] and Haskell []. These functional programming languages are not suitable for developing performance-sensitive NF software, as they all use GCs to reclaim unused memory resources and may incur unexpected program pause during the GC process.

In recent years, C++ [] has evolved into a system programming language that supports a wide range of high-level features that are only seen in functional programming languages. Several promise implementation [] is available in C++.

3. Misc

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.

Figure 1 shows two implementation of a IPS system in StatelessNF. We can see that NetStar has an easier implementation, with smaller number of lines of code, preserved control flow, smaller number of defined functions, and better error handling code.

4. The Asynchronous Flow Abstraction

Figure 2 illustrates our general purpose asynchronous flow abstraction used in NetStar. It is designed to process and inspect connection-oriented flows. It consists of four pluggable part. The client side inspection module aims to mimics the client protocol state. The client side asynchronous loop executes various asynchronous operations. The server side inspection module mimics the server side protocol state while the server side asynchronous loop executes various asynchronous operations as well. The server side asynchronous loop is capable of reconstructing the protocol payload (TCP payload) and analyzing the TCP pay load.

The four parts are pluggable. Programmer can disable uninterested part, making it fully modular. It is designed by combining the Seastar server side programming interface with mOS TCP/IP monitoring sock.

```
class flow_context{
  async_flow<TCPType> _af;
  net::packt _cur_pkt;
  future<> run(){
    _af.on_new_packet().then([]{
      _cur_pkt = _af.get_packet();
      if(!_cur_pkt){
        return make_ready_future<>();
      }
      return mica_ready(flow_key);
    }).then([this](mica_query_response
        automaton state){
      auto new_state = automaton.check(_cur_pkt
          , state);
      if(new_state == alarm){
        drop(_cur_pkt);
        return make_ready_future<>();
      }
      else{
        return mica_write(flow_key, new_state)
            ;
      }
    }).then([](mica_query_response){
      return run();
    });
  }
}
```

(a) IPS implementation based on NetStar.

```
void on_first_pkt(context* ctx, packet* pkt){
  flow_tuple tp = extract_flow_tuple(pkt);
  automaton_state a_st = init_automaton_state();

  async_write_db_with_cb(ctx, tp, a_st,
      subsequent_pkt_read);
}

void subsequent_pkt_read(context* ctx, packet* pkt){
  flow_tuple tp = extract_flow_tuple(pkt);
  async_read_db_with_cb(ctx, tp, subsequent_pkt_write
      );
}

void subsequent_pkt_write(context* ctx, packet* pkt,
    automaton_state* a_st){
  char* next_byte = get_next_payload_byte(pkt);

  while(next_byte != nullptr){
    a_st = process_with_automaton(next_byte, a_st);
  }

  async_write_db_with_cb(ctx, tp, a_st, call_send_pkt
      );
}

void call_send_pkt(packet* pkt, context* ctx){
  send_pkt(pkt);
  update_ctx(ctx, subsequent_pkt_read);
}
```

(b) A callback based implementation for stateless IPS.

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

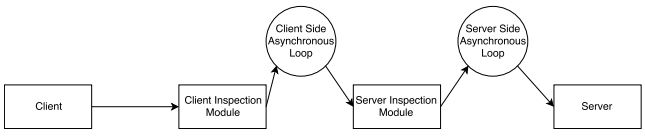

Figure 2: The general-purpose asynchronous flow abstraction used in NetStar.

References

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

[2] HAProxy. `https://www.snort.org/`.

[3] Seastar Project. `http://www.seastar-project.org/`.

[4] Snort intrusion detection system. `https://www.snort.org/`.

[5] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park. mos: A reusable networking stack for flow monitoring middleboxes. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 113–129, Boston, MA, 2017. USENIX Association. ISBN 978-1-931971-37-9. URL `https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/jamshed`.

[6] 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`.

[7] 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.

[8] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 213–226. ACM, 2015.