A Case for a Stateful Middlebox Networking Stack

Muhammad Jamshed, Donghwi Kim, YoungGyoun Moon,
Dongsu Han, and KyoungSoo Park
Department of Electrical Engineering, KAIST

1. INTRODUCTION

Statful middleboxes such as intrustion detection systems, application-layer firewalls, and protocol analyzers are increasingly popular as they perform critical operations in modern networks. Such middleboxes typically operate by maintaining flow states of live TCP connections that pass through a network.

Despite its growing demand, developing a stateful middlebox remains a challenging task. The root of complexity stems from a lack of common programming abstraction for middleboxes that clearly separates flow management from custom middlebox logic. As a result, middlebox developers often resort to writing a complex flow management module from scratch, which results in tens of thousands of code lines that are hardly portable [4, 2]. This is in stark contrast to developing networking applications for end nodes, which significantly benefits from a nice network abstraction layer such as Berkeley socket API. The lack of a reusable networking stack for middleboxes makes the code highly dependent on a custom packet library, which greatly reduces readability, modularity, and extensibility.

In this work, we formulate a general-purpose flow management stack that serves the basic requirements of *monitoring* middlebox applications. The flow management stack should satisfy three requirements. First, it should abstract TCP state management such that the developers can focus on the custom middlebox processing instead of dealing with low level detail. Clear separation of flow management and a custom middlebox logic is the key to high code readability and modularity. Second, it should be flexible enough to express any packet or flow-level event that triggers a special middlebox operation. An event can be as simple as TCP state change or packet retransmission. Or one should be able to extend a basic event by evaluating an arbitrary function (e.g., retransmitted packet whose content is different from the original packet). A flexible event-based system enables a developer to write a middlebox application as a set of logical middlebox events and their event handlers. Third, the flow management stack should allow efficient resource management. Depending on the needs of a middlebox, a developer might want to avoid the flow reassembly on the server side while she actively monitors the flow content from the client side. Or one should be able to deallocate all resources for a flow even if the flow has not terminated yet.

SIGCOMM'15, August 17–21, 2015, London, United Kingdom.

ACM ISBN 978-1-4503-3542-3/15/08..

DOI: http://dx.doi.org/10.1145/2785956.2789999

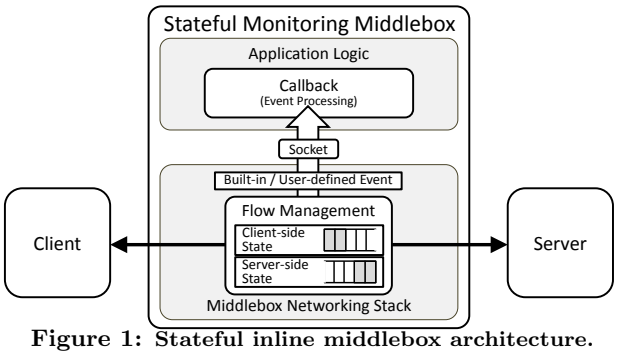

Figure 1: Stateful inline middlebox architecture.

Based on the requirements, we build our networking API for stateful middleboxes based on mTCP [3], a highly-scalable user-level TCP implementation. The mTCP stack serves as the underlying flow management module while we place a number of hooks in flow processing to support basic and extended events. The resulting implementation enables modular middlebox development with high performance.

2. OPERATION MODEL

Figure 1 provides an overview of our middlebox networking stack architecture. The networking stack monitors TCP flows passing through the middlebox, and exposes a socket that abstracts each flow to the application. The application developer defines a set of middlebox events and registers event handlers (or callbacks) that perform custom middlebox logic. When an event handler is called, it is passed a socket for the flow that raised the event, and the application can probe the flow state through our middlebox socket API.

For flow management in the middlebox, each TCP flow context should track both client- and server-side TCP states. This is because a middlebox could act on either the client- or server-side state or both. For example, an IDS could monitor a client-side TCP state transition (e.g., `ESTABLISHED` to `FIN_WAIT_1`) or inspect the data from a server. For efficient resource usage, our stack allows the developer to disable the state management of either side on a per-flow basis.

An event is the critical abstraction in our networking stack, which represents a condition for specific middlebox operation. An event can be either a built-in or a user-defined event (UDE). A built-in event is raised by the flow management module when a certain pre-defined condition is met. For example, a built-in event is generated when a new connection is established or torn down, new data is available in a socket buffer, the TCP state is changed, a new packet arrives, or a packet is retransmitted. A developer can register a handler for a built-in event or can define her own UDE based on a built-in event. That is, a new event can be defined by a built-in event and a custom function that further determines the condition for the event. In short, the user can express a variety of custom events for middlebox processing without dealing with any low level detail.

3. SAMPLE CODE

```
 1 static void // callback for SYN retransmission
 2 on_syn_rexmit(ctx_t ctx, int sock, int side,
 3               event_t e) {
 4       static uint64_t syn_retx = 0;
 5       syn_retx++;
 6 }
 7
 8 static bool // user-defined SYN filter
 9 ft_syn(ctx_t ctx, int sock, int side,
10        event_t e, event_data_t data) {
11       struct pkt_ctx p;
12       // get the current packet context
13       mtcp_cb_getcurpkt(ctx, &p);
14       return (p.tcph->syn && !p.tcph->ack);
15 }
16
..................................................
17
18 // create an UDE for SYN retransmission
19 // it's defined as ON_REXMIT and ft_syn
20 ude_syn = mtcp_define_ude(ON_REXMIT, ft_syn);
21
22 // open a monitoring socket
23 msock = mtcp_socket(ctx, AF_INET,
24                SOCK_MONITOR_STREAM_PASSIVE, 0);
25
26 // register a callback for ude_syn
27 mtcp_register_callback(ctx, msock, ude_syn,
28                        PRE_TCP, on_syn_rexmit);
```

Listing 1: Sample code that counts # of TCP SYN retransmissions (error checking is omitted)

Listing 1 shows sample code that counts the number of TCP SYN retransmissions for the flows that traverse the middlebox. Due to the space constraint, we explain only the main logic of the code. Line 1 to line 15 show how a callback function and a filter function are defined, which are then used in the later code. In line 20, the code defines a user-defined event (`ude_syn`) that detects retransmitted SYN packets. The UDE is based on a built-in event, `ON_REXMIT`, which is raised by the system when a packet in a TCP flow is retransmitted. `ude_syn` extends the built-in event by a filter function, `ft_syn()`, that determines if the retransmitted packet is a SYN packet. If the filter function returns TRUE registered callback function is invoked to process the event. In line 23, a generic monitoring socket (`msock`) is created. By default, it monitors all TCP flows passing through the middlebox unless the monitoring scope is limited by `mtcp_bind_monitor()`. In line 27, a callback function (`on_syn_rexmit`) is registered for `ude_syn` with the montoring socket. `PRE_TCP` in the function parameter requires that the event should be raised before updating the middlebox TCP state for the flow.

4. PRELIMINARY EVALUATION

We are currently porting popular middlebox applications to our framework. in The preliminary results look promising.

Scalability & Composability: We evaluate the robustness of our system in a testbed that consists of four sets of web client/server machines and an inline network toy web monitor (an Intel Xeon E5-2697 14-core machine with 64 GB RAM) that counts the number of ongoing connections. We test the performance under two scenarios. One monitoring socket is tested without TCP management while the other is used to monitor flows without payload inspection (per-packet processing = without TCP semantics; per-flow processing = TCP sockets without deep payload inspection capability). As the Figure 2 evaluation graph suggests, our toy monitor

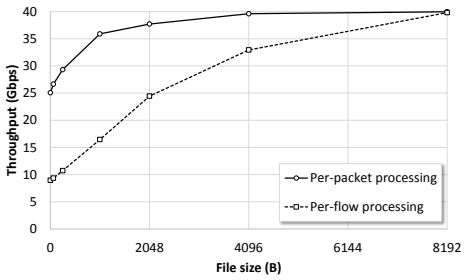

Figure 2: Monitor Performance (`HTTP GET` 16K reqs/s)

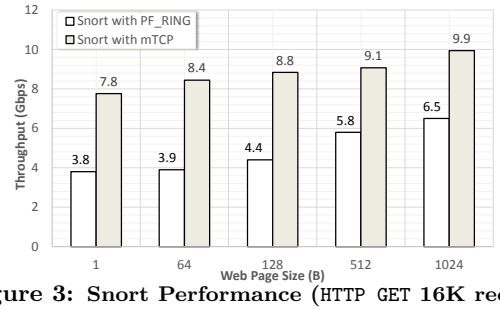

Figure 3: Snort Performance (`HTTP GET` 16K reqs/s)

can withstand analyzing rates of up to 40 Gbps for 8KB web pages while it can manage flows at a rate of at least 10 Gbps for small file sizes.

Improved abstraction: We have partially ported Snort [4], a popular open-source signature-based network security monitor. We replace around 8,000 lines of source code with only 799 lines to support Snort's default TCP stream management module. At the moment, our version skips some elaborate checks such as segment overlapping and connection anomaly checks (*e.g.* arrival of TCP SYN packet on an already established connection). Our porting process has substantially simplfied the core framework since the new code focuses on pattern matching rather than flow management. We evaluate the performance of our stack by generating innocent client/server web request/response pages for varying file sizes in the same testbed with around 3,000 HTTP Snort attack signatures. For comparison, we also perform the same test with default Snort setup with PF_RING [1] I/O driver. Figure 3 shows that our version outperforms the I/O-optimized Snort by a factor of 1.5 to 2.2.

Enhanced monitoring capabilities: Some security monitors (such as Snort and Suricata) analyze flows against elaborate reassembling routines for examining payload spanning several successive TCP segments. These routines are usually employed to detect attack patterns in malicious flows. Our stack provides user-friendly APIs that make development of such routines much simpler in practice.

5. ACKNOWLEDGEMENT

This work was supported by the ICT R&D program of MSIP/IITP, Republic of Korea [14-911-05-001, Development of an NFV-inspired networked switch and an operating system for multi-middlebox services].

6. REFERENCES

[1] PF_RING I/O Driver. http://www.ntop.org/products/pf_ring/.
[2] Suricata Open Source IDS. http://suricata-ids.org/.
[3] E. Jeong and et al. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *NSDI*, 2014.
[4] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *LISA*, 1999.