mOS: An open middlebox platform with programmable network stacks

Shinae Woo

ABSTRACT

Though the growing popularity of software-based middleboxes raises new requirements for network stack functionality, existing network stack have fundamental challenges in supporting the development of high-performance middlebox applications in a fast and flexible manner.

In this work, we design and implement an enriched, programmable, and extensible network stack and its API to support the various requirements of middlebox applications. mOS supports proxy and monitoring function as well as traditional end TCP stack function. Further, we allow applications extend TCP functionality by hooking in middle of TCP processing and define user-level events on TCP state. Meanwhile, `Epoll`-like API allows applications manipulate read-/write from/to byte stream buffers in an efficient way.

To support an efficient consolidation of multiple middlebox applications in a single machine, mOS will allow multiple middlebox applications share the same TCP processing context without duplicated IP/TCP processing.

We show that mOS can support various middlebox applications in an easy and efficient way without building TCP functionality from scratch.

1. INTRODUCTION

Software-based middleboxes are growing popular with their flexibility and ease of extensibility [2, 7]. The movement towards software-based middleboxes raises new requirements for network stack functionality. Existing network stack, however, introduces fundamental challenges in supporting the development of high-performance middleboxes in a fast and flexible manner.

Middlebox applications require different network functionality from existing stacks. Middleboxes need to monitor flow context such as extracting flow-level information from packet streams, detecting retransmission, tracking flow lifetime. Some other middleboxes require to implement proxy functionality over traditional TCP sockets to support load balancing, WAN optimization, or caching. However, such functionality is not provided by traditional TCP stacks.

Many middlebox applications implement their own TCP functionality from scratch. Many middlebox requires only partial functionality from whole TCP stack functionality. For example, stateful NAT or stateful firewall needs to track TCP lifetime to preserve the traffic state to serve. To detect malicious retransmission, it needs to track reordered buffer contexts to detect retransmitted packets and compare their contents [9] to detect maliciousness. Proxy is one of common functionality in middleboxes, but developers implement complex proxy functionality on top of traditional TCP stacks.

Middlebox consolidation provides a way to efficiently manage multiple middleboxes in a network. For example, CoMB[12] showed the benefit of network protocol sharing (e.g., IP, TCP, HTTP, FTP) between multiple applications. However, current network stack does not yet support for protocol sharing or middlebox consolidation.

In this paper, we propose mOS, an open middlebox platform with programmable network stacks. mOS support various requirements of different middleboxes. It has monitoring and proxy functionality as well as traditional TCP stack functionality. Application can selectively choose their desire functionality such as tracking TCP lifetime or reordering byte stream to monitoring flows. On top of our new stack, we design a new set of API to easily develop new middleboxes on mOS.

We also allow applications hook TCP processing with specified events to efficiently extend TCP monitoring functionality. Some middleboxes may be interested in specific TCP states such as retransmission, long RTT or spurious time out events. Rather than implementing a specialized TCP stack which supports ample set of TCP events, we choose to let application developers extend TCP stack to implement their specific requirements by hooking on TCP processing using built-in events or user-defined events.

We implement mOS on top of mTCP [11], a high performance user-level TCP stack. We modify mTCP to support multiple applications at the same time. We add monitoring and proxy TCP stack by refactoring mTCP stack. Over the top of mOS, we will implement various kinds of applications to prove the feasibility and extensibilty of mOS.

Then this paper makes the following contributions:

- It identifies new requirements of middlbox applications over TCP stack functionality.
- It designs an extensible and flexible TCP stack API to meet the identified requirements.
- It implements mOS including new network stack functionality to support ease of development middlebox applications.
- It shows the feasibility of mOS by implementing middlebox applications using our API.

The remain of this paper is organized as follows. Firstly, we show the challenges to support software-based middlebox applications in Section 2, and explain our design principles in Section 3. Then we will cover mOS architecture in Section 4) and its API in Senction 5. Section 6 shows mOS's implementation detail and we evaluate mOS and applications in Section 7. We conclude it in Section 8.

2. CHALLENGES FOR SOFTWARE-BASED MIDDLEBOX APPLICATIONS

Software-based middleboxes are expected to develop in a fast and flexible way. However, lack of support for sufficient network functionality is an obstacle in middlebox implementation. Middlebox developers should develop network functionality as required before implementing their own application-layer logic [10, 13, 9]. In this section, we identify the challenges of implementing software-based middleboxes in detail.

Exposing flow-level information: Traditional network stacks lack support for exposing flow-level information. Moreover, each middlebox application is interested in different types of information. For example, IDS requires reading in-ordered TCP byte stream for bytestream inspection, whereas an accounting system only needs to know payload length to calculate the amount of traffic served to each user.

No existing tool or stack can effectively provide such information. `TCP_INFO` provides TCP-level information by `getsockopt()` system call[4]. But the provided information is limited to `TCP_INFO` structure. Many system calls also bring a burden on system.

Lack of in-line TCP functionality: Existing network stacks do not support in-line TCP functionality, which is a common requirement for middlebox applications. Middleboxes usually work in the middle of end-to-end flow stream for the role of packet stream monitoring, cutting some flows in the middle, or intercepting a connection in the middle and breaking it into two.

Middleboxes often need to reintegrate flow-level context from packet streams. While existing networking stacks actively participate in creating and controlling flow-level contexts (e.g., slow start threshold, congestion window size, bytestreams to send), some middleboxes are minimally involved in controlling them. More importantly, they need to extract a set of interested flow-level information from packet streams. To achieve this, middlebox application developers build their own networking stack implementation, which is a subset of full network stack functionality [10, 13, 9].

Proxy is a common middlebox functionality which serves flows in the middle of end-to-end context. (e.g., WAN accelerator, Web caching). Each proxy server independently implements the common connection management process in the middle of connections[1, 3, 6]. Their job includes (a) choosing target connections (b) accepting a connection from client (c) creating a new connection for the other side, and (d) managing TCP states to tear down or handle exceptional cases. Even though connection management is an error-prone and tedious process, developers need to implement the same logic for every proxy application.

Diverse range of requirements: Different applications impose different requirements on networking stacks. Some applications require only partial functionality of TCP such as monitoring 3-way handshake, whereas others need to extend the basic functionality to meet their own purpose. For example, a stateful firewall may be interested only in tracking flow life time (e.g, NetFileter [5]). A security system seeks to expand flow management functionality to detect malicious retransmission (e.g., Abacus [9]). Nevertheless, no existing network stack is capable of supporting diverse requirements with standard and general APIs.

Sharing TCP context: Sharing TCP context between middlebox application can save the heavy operational overhead for TCP processing. For example, running firewall and webserver in a single machine together while sharing same TCP context will save the additional TCP management operation cost. In Linux, NetFilter [5] enables to run firewall in addition to normal TCP applications, but it independently manage TCP context using TCP conntrack from Linux TCP stack.

There is no open platform for sharing common middlebox functionality or for adding new functionality to an existing middlebox server. xOMB proposes a programmable middlebox platform while limiting the processing model to a protocol acceleration proxy that cuts one flow into two [8]. On the other hand, CoMb shows the benefit of consolidating multiple middlebox applications into a single machine [12]. However, they lacks flexibility such as passive connection monitoring that does not require flow halving.

3. DESIGN CONSIDERATIONS

The challenges introduced in Section 2 make it difficult to rapidly develop middlebox applications. Implementing TCP functionality usually takes effort because of the inherent complexity of their logic. Also, it imposes redundant cost on many middlebox developers. Therefore, we will design and implement mOS, which includes efficient and programmable network stacks. We believe that mOS allows application developers to focus on their own logic over simple and general APIs by reducing the cost of tedious network stack implementation.

The new stack architecture includes 3 TCP components as shown in Figure 1. mOS will provide new TCP stack functionality for monitoring and proxy as well as traditional end TCP stack functionality. In all cases, applications can flexibly monitor and extend TCP stack functionality. In this section, we will explain important design consideration to support described requirements.

3.1 Unified flow management

mOS enables application developers to selectively choose TCP features by well-defined API from a set of predefined features. mOS provides in-middle network stack functionality as well as existing end-node stack as shown in Figure 1. Application can choose any of this functionality based on its role and location of network. End-node socket and proxy socket can be used exclusively together. It means that no single TCP context can be managed in both sockets. Meanwhile, monitoring socket can be used together with other sockets.

mOS should provide unified flow management, if a single TCP context can included more than one socket (e.g., multiple monitoring sockets or a monitoring socket and a end-TCP socket). It will only process TCP context single time, and reuse the same TCP context to multiple applications.

Different monitoring applications may require different level of TCP management. For example, an application may choose to track TCP lifetime only, not to manage byte stream. To efficiently support this various requirements, mOS should provide selective API to minimal TCP processing. If multiple monitoring applications sharing same TCP context require different level of TCP management, mOS will automatically choose minimal union of TCP processing to support all monitoring applications.

3.2 Events-based API

In mOS, middlebox programming can be done by writing actions in response to a diverse set of flow-level ”events” such as flow creation, new byte stream to read, or new packet arrival. mOS will allow two kinds of event-based API - one is callback-based API,

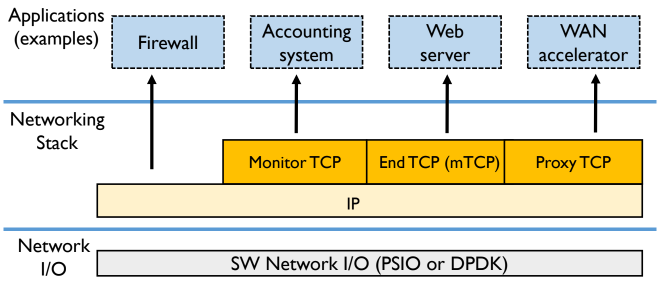


Figure 1: Architecture of mOS

and the other is `epoll`-like event-driven API. Both API consume the produced events from TCP stack.

Application developers associate callbacks to produces events and this callback will be executed in the middle of TCP processing on behalf of applications. Using callback API, application can inspect TCP context or extend functionality by additional processing such as retransmission detection, or packet drop.

While callback provides efficiently and synchronously inspect TCP context, `epoll`-like event-driven API will be useful for efficient buffer access for `read()` or `write()`. Unlike callback, `epoll`-like API will aggregate multiple events into single events. While it can not preserve the moments of events happened, it effectively summarize the multiple events to the application side so that the communication overhead between applications and mOS will be minimized.

3.3 User-defined events

In addition to predefined events in mOS, it also allows users to define their own events related to changes in TCP contexts and byte streams. User-defined events are an important design decision that allows efficient extension of the basic TCP functionality to meet various requirements of middlebox applications.

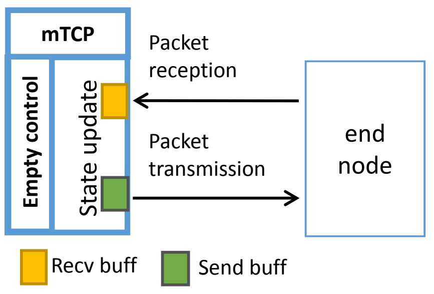
As an example, malicious retransmission can be simply detected without building a complex TCP module from scratch [9]. It only requires defining an retransmission event based on TCP and packet context, and adding a callback for the custom events to check whether it is malicious.

3.4 User-friendly API

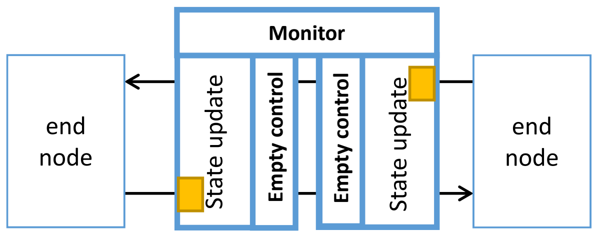
Even mOS provides new functionality which was not exist in previously, our provided API should be easy to use without forcing application developers to learn many new concepts. Therefore, we choose to follow the syntax of existing BSD-socket API and `epoll` API. We minimally modify and extend the existing API to support mOS's additional functionality.

Socket type	Target Application	Functionality
End-node TCP socket	lighttpd, memcached	Traditional end-node TCP functionality
Monitor TCP socket	PRAIDS, MonBot, Abacus, Snort	Inspecting TCP connection states and byte streams in middle
Proxy TCP socket	Load balancer, WAN optimizer, SSLShader	Intercepting a single TCP connection and break down into two connections

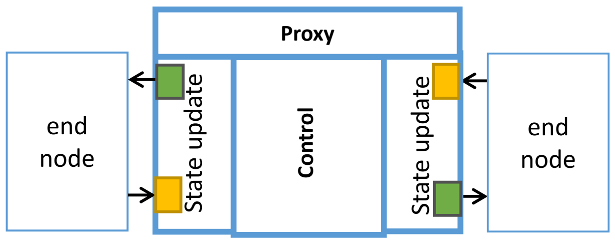
Table 1: Sockets provided in mOS



(a) End-node TCP socket



(b) Monitor socket



(c) Proxy socket

Figure 2: Comparison between end-node TCP, monitor TCP, and proxy TCP

4. ARCHITECTURE

In this section, we will explain the architecture of mOS network stack. mOS supports additional socket interfaces to abstract monitoring and proxying functionality and associated functionality. Every communication between applications and mOS is initiated based on events. We will explain more details in following subsections.

4.1 mOS socket

mOS provides specialized sockets for monitoring and proxying functionality. mTCP supports TCP socket interfaces of `MOS_SOCK_STREAM` type for traditional end-node TCP connections. In addition to this, mOS also supports `MOS_SOCK_MONITOR` and `MOS_SOCK_PROXY` type sockets each for monitoring and proxy.

These two new socket type provides different abstraction for TCP connections. The `MOS_SOCK_STREAM` type socket abstracts a uni-directional TCP connection which has a single receive buffer and a single send buffer. However, monitoring and proxy socket provides a bi-directional TCP connection in middle of network link with each direction for server-side or client-side TCP connection abstraction.

Figure 2 shows the graphical representation of different TCP socket abstraction. Traditional end-node TCP take charge of reliable byte stream transmission with the other side of end node. To achieve reliable transmission and fair share from other TCP flows sharing links, it controls the transmission using various techniques such as flow control, congestion control, and Naggles algorithm, etc.

Monitor and Proxy socket shows different functionality from end-node TCP sockets. Monitor sockets allows passively monitor the TCP flow states and/or bytes stream buffers in best effort way. It can monitor flows in middle of links as well as flows which are served by end-node sockets.

Proxy socket allows application actively intercept a single connection to break down into two connection. In middle of two end nodes, it serves the byte streams as a server to the client, and as a client to the server. Proxy socket can be used to Web cache, load balancer, or SSL proxy.

Socket itself is an abstraction for network stack functionality, and it is independent from its internal implementation. For example, a monitoring socket can be used to abstract flow contexts in middle of network link or flow contexts shared with end-node TCP context. Application developer does not need to know whether the flow context is reconstructed from packet stream in middle or is shared with end-node TCP stack. mOS will choose an adequate TCP processing to provide enough TCP context for applications.

Built-in events name	Event source	Explanation
MOS_ON_PKT_IN	Packet	Packet reception event
MOS_ON_PKT_OUT	Per-packet	Packet transmission event
MOS_ON_CONN_SETUP	Connection	Flow creation event
MOS_ON_CONN_TEARDOWN	Connection	Flow destroy event
MOS_ON_CONN_NEW_DATA	Send or receiver buffer on connection	New data is ready to read in socket buffer
MOS_ON_ERROR	Packet or Connection	An Error happens during flow processing

Table 2: Built-in events for callback

User-defined events	Base-line event	Explanation
MOS_ON_SYN	MOS_ON_PKT_IN	SYN packets
MOS_ON_PKT_RTX	MOS_ON_PKT_IN	Packet reception event
MOS_ON_HTTP	MOS_ON_CONN_NEW_DATA	Byte stream start with a ”HTTP” keyword
MOS_ON_FTP	MOS_ON_CONN_NEW_DATA	Byte stream start with a ”FTP” keyword

Table 3: Example of user-defined events

Built-in events name	Socket	Explanation
MOS_EPOLL_IN	End-TCP socket	Available to read from buffer
MOS_EPOLL_OUT	End-TCP socket	Available to write to buffer
MOS_EPOLL_IN_CLT	Proxy socket, Monitor socket	Available to read from client buffer
MOS_EPOLL_IN_SVR	Proxy socket, Monitor socket	Available to read from server buffer
MOS_EPOLL_OUT_CLT	Proxy socket	Available to write to client buffer
MOS_EPOLL_OUT_SVR	Proxy socket	Available to write to server buffer

Table 4: Events for Event-poll API

4.2 Middlebox TCP Stacks

In this subsection, I will explain detailed TCP stack functionalities. These TCP stack Implementation is still in progress. I will specify the implementation progress detail in Section 6.

4.2.1 Monitoring Stacks

If an application create a monitoring socket, and the associated flow context are already managed by end TCP socket or proxy TCP socket, then the monitor socket will serve the associated monitor processing using the already created flow context. However, if no end TCP stack or proxy stack is charge of managing associated TCP flow context for monitoring, then monitoring stack itself will provide a TCP context for the stack.

Monitoring stack reconstruct the TCP context from the packet stream, while it minimally involves any control mechanism for the flow between two end-node. Specifically, it manages TCP context by tracking the TCP state transition diagram from the packet stream, reordering byte stream from the data packets, and flushing byte stream from the buffer from ack packets.

This monitoring stack will provide TCP inspection in best-effort way since it cannot control the rate of transmission or TCP context. In case of end-TCP stack, it can control the rate of TCP context using flow control or congestion control. But monitoring stack only allow to passively monitor without controlling. Moreover, depending on the location of middlebox, it may experience packet loss so that monitoring byte stream may not available. Packets can be lost during mirroring or tapping in switches. Therefore, there is no assurance that monitoring stack can monitor every TCP contexts and every byte stream.

Whenever mOS detects failures on TCP management, it raises errors to applications. Slow applications may make mOS processing slow down or overload finite size of buffers. Or it may detect packet losses. Then application can choose to stop the monitoring or continue the monitoring even with lost TCP context.

Monitoring stack allows 3-levels of monitoring. Fist level is reconstruct both TCP context and byte stream. Application can read byte stream from both transmitted directions. It is useful for byte stream monitoring application such as Intrusion Detection System (IDS). Second level is reconstruct TCP context and byte stream context, but not byte stream reordering. Some applications may not require to read byte stream, but may require to know byte stream context such as how many bytes are passed. TCP-level accounting system needs

API name	Explanation
`mtcp_socket()`	Create new socket for monitoring listener and proxy listener
`mtcp_bind_connfilter()`	Bind a connection filter to a socket Replacement with `bind()` in end-TCP

`mtcp_activate_monitor()`
`mtcp_activate_proxy()`
	Activate monitor or proxy socket
	Replacement with `listen()` in end-TCP

`mtcp_accept()`	Accept new monitor connection or proxy connection from monitoring listener or proxy listener

`mtcp_read()`
`mtcp_write()`
	Read or write byte stream

`mtcp_close()`	Close existing socket

Table 5: mOS socket API

this level of functionality. Last level is reconstruct TCP context only. Stateful firewall or network address translation (NAT) need to track life time of TCP. Applications can choose desired management level.

4.2.2 Proxy Stacks

Proxy is widely used functionality in middlebox applications such as web caches or media caches. Currently, most of proxy is implemented over top of end-node TCP stack. Applications use `accept()` and `connect()` to intercept connections and manage many corner cases on both direction of connections. For example, if one direction of connection is closed with errors, the paired direction will be closed accordingly.

mOS will provide simple abstraction for proxy functionality. It simplify complex processing with simple API. For example, proxy stack intercepts the connections in middle of link. it accepts the connection from the clients, and connects to the server side. It relays the byte streams between clients and server. It also hides complex corner cases in paired connection management. Using proxy stack, application developers easily implement proxy application just with simply `read()` and `write()` byte streams each other.

4.2.3 Monitoring with end-node or proxy stack

mOS can monitor the TCP connections which is managed by end-node or proxy stacks. In this cases, monitoring socket reuses the TCP contexts which is processed by end-node TCP or proxy stacks. Meanwhile the monitoring should not affect the performance of end-node TCP or proxy socket.

`read()` and `write()` byte streams in monitoring applications is provided in best-effort way. If monitoring applications read much slowly than end-node TCP or proxy applications, monitoring applications may lost some byte streams.

4.3 Events

mOS TCP stack produces various kinds of events as flow processing results. These events are consumed by applications to inspect TCP states, extend TCP functionalities, or efficiently manipulating byte stream buffers of TCP flows. TCP stack produces two sets of events for callback and event-driven each and they are represented in Table 2 and Table 4.

mOS will provide minimal built-in events for callback as described in Table 2. We intentionally choose minimal set of built-in events rather than implement an ample set of built-in events including derived events such as retransmission event or time out event. If an application needs more than built-in events, it will create a user-defined event (UDE) based on built-in events and TCP context.

Applications can flexibly extend TCP functionality using user-defined events. Table 3 shows usage examples of callback and UDE. On the packet reception events (`MOS_ON_PKT_IN`), we can define new events by inspecting the type of packets (`MOS_ON_SYN`) or TCP context (`MOS_ON_RTX`). On an event for indicating new data available (`MOS_ON_CONN_NEW_DATA`), we can decide whether it uses our interested protocol such as HTTP, FTP by inspecting the first byte stream of the flows (`MOS_ON_HTTP`, `MOS_ON_FTP`).

5. mOS API

mOS should provide simple and flexible API to application developers. In this section, we will explain how we design mOS API and it's usage examples as well.

5.1 BSD-like Socket API

Most of our API follows Linux BSD-socket API like in Table 5 and minimally modify or add API to adequately provide new functionalities. `mtcp_` prefix is attached with every API since our platform is extended over mTCP.

We extend `socket()` function call to create monitor-

Type	API name	Explanation
Callback	`mos_define_ude()`	Define new user-defined event
Callback	`mos_undefine_ude()`	Undefine exisiting user-defined event
Callback	`mos_register_callback()`	Register callback function on event
Callback	`mos_unregister_callback()`	Unregister callback function
Event poll	`mtcp_epoll_create()`	Create epoll structure
Event poll	`mtcp_epoll_ctl()`	Add or delete event on epoll structure
Event poll	`mtcp_epoll_wait()`	Wait until new event arrived

Table 6: Event manipulate API provided in mOS

Criteria	Epoll	Callback
Running on	Application thread	mOS thread
Goal	Buffer access (`read()`, `write()`)	Peeking TCP context
Input	socket id	`struct flow_ctx`, `struct pkt_ctx`
Events	`EPOLLIN_*`, `EPOLLOUT_*`	`ON_PKT_*`, `ON_CONN_*`
User-defined events	Not allowed	Allowed

Table 7: Difference between epoll and callback API

ing listening socket or proxy listening socket. To create a end-node TCP listening socket, application developers use `MOS_SOCK_STREAM` as a parameter. mOS allow simple create a monitoring listening socket and proxy listening socket, by using `MOS_SOCK_MONITOR_LISTEN` and `MOS_SOCK_PROXY_LISTEN` paremeters each.

`bind()` cannot be used in monitor and proxy TCP sockets directly. The original parameter `const struct sockaddr` in `bind()` is only represent single end host. However, middlebox applications need to represent a pair of two end hosts with more flexible way such as subnet or IP range. `bind_connfilter()` binds a connection filter to a monitor listening socket or a proxy listening socket. Connection filters provide more flexible way to bind a pair of addresses to a socket such as subnet or IP range.

mOS provides activation API for middlebox listening socket such as `activate_monitor()` and `activate_proxy()`. With this function call, mOS will start processing of monitor or proxy. We do not use `listen()` call, since the middlebox semantic is different from end-node TCP.

The rest of socket function calls such as `accept()`, `read()`, `write()`, and `close()` are same with traditional end-TCP. Maintaining the similar API with widely used BSD socket API, application developers easily use new functionality without learning much.

5.2 Event handling API

mOS provides two set of event-driven APIs. One is event-polling based API and the other is callback-based API. The conceptual representation of mOS is represented in Figure 3. Firstly, event-polling based API is mostly used to manage TCP context such as

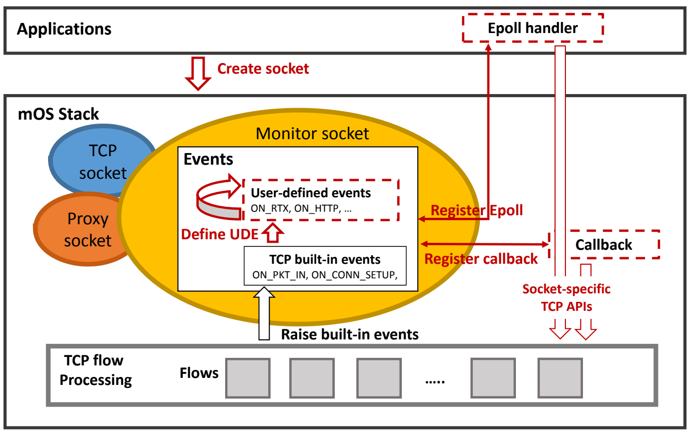


Figure 3: Conceptual representation of mOS API

`accept()` or `close()` them, and access byte streams such as `read()` or `write()` from/to them. Secondly, callback-based API is mostly used to inspect TCP context or extend TCP functionality such as detecting retransmission event or time out event.

The differences between epoll API and callback API are described in Table 7. Epoll function calls are run on application thread context, and callback function calls are run on mOS thread context. They use different set of API calls with different parameters. For example, epoll APIs use socket id to access the flow context. However, callback API use flow context and packet context rather than socket id. Using this context, application can monitor the internal state of flow or packet context at the moment of event raising. Event-driven and Callback API use its own set of events as described in Table 4 and Table 2. Lastly, only the built-in events for callback can be extended to user-defined events.

5.2.1 *Event-polling (*`epoll`*) API*

Event-polling API (`epoll`) is one of widely used API to I/O event notification. `EPOLL_IN` and `EPOLL_OUT` event are indicating each for read and write availability of byte streams. Last 3 columns in Table 6 show the epoll API in mOS. Excepts the `mTCP_` prefix, mOS provides exactly same epoll API with original API. However, we extend the epoll events to indicates that whether the I/O events are associated with server-side or client-side buffers using `SVR` and `SVR` postfix.

Figure 4 shows an example usage of epoll API for monitor socket. Firstly, it creates a monitor socket, binds them with connection filter and activates it. We use `mtcp_epoll_wait()` to receive events from monitor listening socket. Whenever new monitor connection is available, it will receive `EPOLLIN` event from monitor listening socket. Whenever new byte stream is available, it will receive `EPOLLIN_SVR` or `EPOLLIN_CLT` events. Then application can read the associated byte stream using `mtcp_read()` function.

I intentionally omit an example usage of epoll API for proxy socket, but it is also similar with monitor socket except it will receive also `EPOLLOUT_*` events to write buffers on each side. In this case, mOS APIs hide the complexity of managing a pair of proxy connection using simple abstraction such as `activate_proxy`, simple `read()` / `write()`, and handling corner cases on closing connections.

This function call diagram is very similar with end-TCP node's epoll-call diagram except replaced function names and additional event name to indicate the location of I/O buffer (server-side or client-side). Using this friendly set of APIs, application developers can easily use mOS API.

5.2.2 *Callback API*

mOS enables application hooking TCP processing for monitoring TCP context as well as extending TCP functionality using a simple set of API.

The API for callback is listed in first 4 columns of Table 6. Applications can define their own events based on built-in events. Also, applications can register callback fucntion either on built-in events or user-defined events.

Inside callback functions or user-defined events filter functions, applications can access the flow-level context or packet-level context. Applications can set/get a user-level context for each flow. Following examples from Algorithm 1, 2, 3 show how this API actually be used.

Algorithm 1 shows an example of creating monitor socket for callback. Firstly, it create a simple monitor

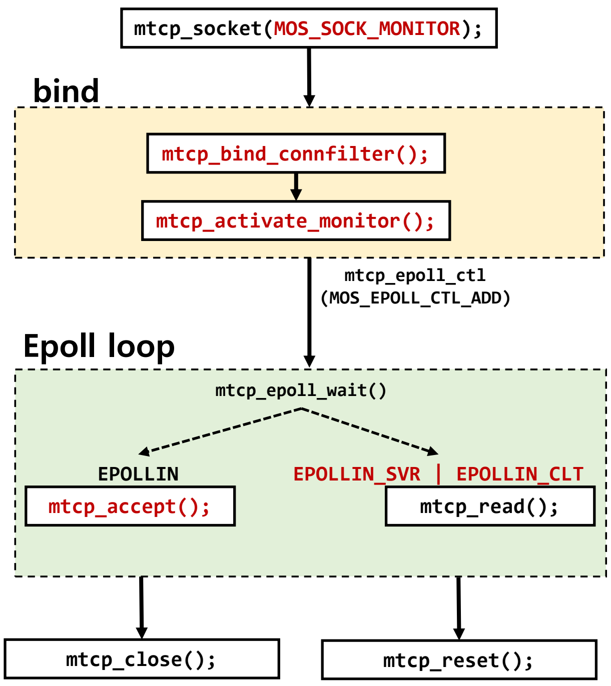


Figure 4: Usage example of epoll on monitor socket

socket (line 1:10). And using the monitor socket, it can add callback function using `mtcp_register_callback()` or define new events using `mos_define_ude()` on the socket. Whenever associate built-in events are raised during TCP processing, mOS will call the registered callback.

Algorithm 2 shows an example of byte stream peeking using callback API. Application will peek byte streams using `mtcp_peek()` (line 16). Unlikely `read()` API, `mtcp_peek()` does not preserve the last read offset. Application need to specify the desired offset and amount of read byte stream size. To save the next offset to read, application may use `mtcp_get/set_uctx()` (line 12:13, 22:23) to preserve the user-level context associated with each flow. Application also peek TCP context by accessing read-only `struct flow_ctx`, which provides associate packet context if available, TCP state, byte stream context, sequence number, packet loss, RTT, etc.

Application can define new user-level event based on built-in event. Algorithm 3 shows an example of defining new user-level event for detecting 'SYN' packets. Application register a filter function to detect SYN packet on packet reception event (`MOS_ON_PKT_IN`) in line 14:16. A filter function returns 1 if the event condition is satisfied, or returns 0. In case of detecting SYN packet, it checks the TCP header in associated packet context in line 1:12.

```
monitor = mtcp_socket(ctx->mctx,
      AF_INET,
      MOS_SOCK_MONITOR_LISTEN,
      0);

if (monitor < 0) {
  TRACE_ERROR("Failed to crete
      monitor socket!\n");
  return -1;
}

mtcp_register_callback(mctx, monitor,
      MOS_ON_CONN_NEW_DATA, MOS_POST_TCP,
      callback_on_conn_new_data);
```

Algorithm 1: Creating monitor socket

```
static void
callback_on_conn_new_data(mctx_t mctx,
        const struct flow_ctx *fctx,
        uint64_t events)
{
#define buf_len 2048;
char buf[buf_len];
int ret = buf_len;
int tid = mtcp_get_thread_id(mctx);

int64_t offset = (int64_t) mtcp_get_uctx (
           mctx, fctx);

while (ret == buf_len) {
  ret = mtcp_peek (mctx, fctx, buf,
           offset, buf_len);
  if (ret > 0)
    offset = offset + ret;
}

mtcp_set_uctx(mctx, fctx,
         (void *) offset);

return;
}
```

Algorithm 2: Register callback

```
static int
ude_on_pkt_syn(mctx_t mctx,
  const struct flow_ctx *fctx,
  uint64_t events)
{
  struct pkt_ctx *pctx = fctx->pctx;
  struct tcphdr *tcph = fctx->pctx->tcph;

  if (tcph->syn)
      return 1;
    return 0;
}

mtcp_define_ude(mctx, monitor,
      MOS_ON_PKT_IN, MOS_POST_TCP,
      ude_on_pkt_syn);
```

Algorithm 3: Define user-defined event

6. IMPLEMENTATION

We are implementing mOS to support middlebox applications over the mTCP [11], a multi-core scalable user-level TCP.

mOS requires refactoring of mTCP architecture to support multiple application simultaneously. mTCP is implemented as a user-level library to be linked from user applications. mTCP exclusively attaches the RX/TX queues on the NICs so that only a single application can link mTCP library in a system. To overcome this problem, we refactored mTCP to run as a service which exclusively manages communication with NICs by RX/TX queues and manages multiple applications on it.

We implement TCP context exposing APIs on mTCP so that application developers can inspect TCP states as they requires. We modify TCP stack implementation of mTCP to expose built-in event and add APIs to register callback and defined UDE.

For monitoring socket, we built a specialized monitor stack by modifying mTCP stack. When a new SYN packet comes, it creates a monitoring connection and tracks TCP context by monitoring associated packets of the connection. Specifically, it tracks the 3-way handshakes establishment, data/ACK packets, and 4-way handshake teardown.

This is a currently ongoing work so that we should complete API such as undefine user-level events and unregister callback. As a future work, we should complete byte stream monitoring on monitor socket, and implement proxy socket functionality on top of end-node TCP stack.

We implement toy applications to check the feasibility of mOS APIs. We only cover the monitoring applications here since proxy functionality is not yet implemented.

Firstly, we implement byte stream monitoring application using monitoring socket and callback APIs. The skeleton code looks like in examples in Algorithm 1 and 2. Secondly, we implement TCP status inspection application for checking TCP retransmission using monitoring socket and user-defined events.

7. EVALUATION

We evaluate the performance of mOS monitoring system. Firstly, we compare the mOS monitoring system with the performance of Linux TCP status monitoring function - TCP_INFO structure using system call. Secondly, we compare the performance of multiple applications with singular application. Thirdly, the performance degradation of registering callback and defining user-defined event are explored.

For this evaluation, we use two machines, each for a client and a server. Each machine is equipped with Intel Xeon CPU E5-2690 Hexacore (8 cores), 128 Gbps memory, and IXGBE 82599 10Gbps NIC.

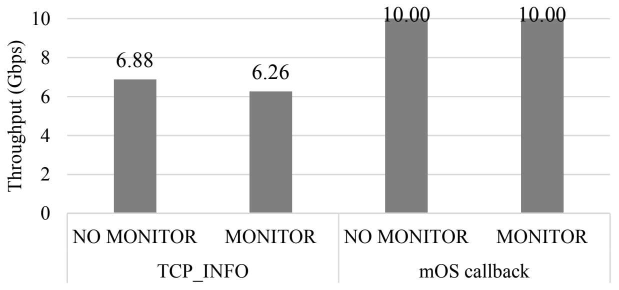


Figure 5: Comparison between TCP_INFO and mOS monitoring

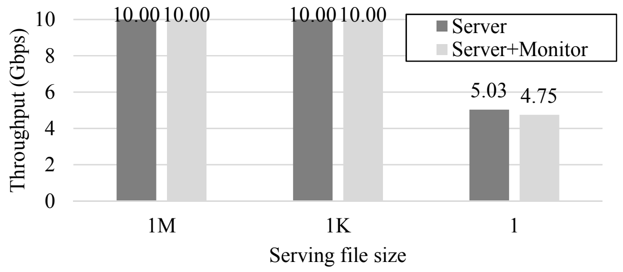


Figure 6: Performance of multiple apps

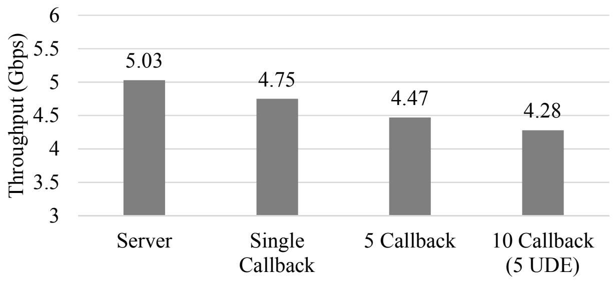


Figure 7: Performance of Callback and UDE

In Linux TCP stack, applications can monitor the TCP context using `getsockopt()` system call. With every system call, it fills the TCP_INFO structure which has 29 elements such as TCP state, the number of retransmission, window size scaling, retransmission timeout, etc. With every system call, it fills all field in TCP_INFO structure and it takes about 250 ns. Application can see a snapshop at the moments of system call, however, it does not know when a retransmission event or a timeout event happens. mOS can monitor the TCP context using system call. This means that mOS callback can preserve the status at the moment of event raising. Also it can inspect the only interested set of features rather than filling full elements in a structure.

Figure 5 shows the performance comparison between TCP_INFO and mOS monitoring with 2,000 concurrency serving 1MB file. They inspected TCP status 10 times per connection. It might not be a fair comparison since mOS is based on high performance user-level TCP stack. However, while TCP_INFO system call monitoring degraded the original TCP stack's throughput, mOS stay the link rate with monitoring.

We compare the performance of running a single application with running two applications same time by sharing TCP context. Firstly, we run a web server with normal end-TCP socket, and add a byte stream monitoring application with monitor socket. Monitor socket and end-TCP socket share the same TCP context together. In Figure 6, running two applications with sharing TCP processing showed similar level of performance with a single application. Of course the level of performance degradation may differ from the application's workload. However, we can see that sharing protocol stack has a benefit from saving additional TCP context management.

Lastly we see that how much overhead induced from registering callback functions and defining user-defined events in Figure 7 with 1B file transfer. Adding a single callback and adding 5 callbacks, the performance droped about 5.5% and 11%, respectively. Adding more 5 UDE filters and corresponding 5 callback functions, the performance droped about 15 %.

8. CONCLUSIONS

In this proposal, we propose mOS, a new network stack to support diverse requirements of software-based middlebox. mOS has extended TCP functionality such as exposure of flow-level information and in-middle TCP functionality. Moreover, mOS enables developers to choose a subset of TCP functionality, or to extend basic functionality for meeting each application's specific requirements. We believe that this programmable network stack will open a new paradigm in developing middlebox applications.

9. REFERENCES

[1] Apache Proxy. `http://httpd.apache.org/docs/current/mod/mod_proxy.html`.
[2] Brocade Vyatta. `http://www.brocade.com/index.page`.
[3] lighttpd Proxy. `http://redmine.lighttpd.net/projects/1/wiki/Docs_ModProxy`.
[4] Linux TCP. `http://linux.die.net/man/7/tcp`.
[5] NetFilter. `http://www.netfilter.org/`.
[6] Nginx Reverse Proxy. `http://nginx.com/resources/admin-guide/reverse-proxy/`.
[7] SilverPeak VX Software. `http://www.silver-peak.com/products-solutions/wan-optimization/vx-software`.
[8] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xOMB: Extensible Open Middleboxes with Commodity Servers. In *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems (ANCS)*, 2012.

[9] Y. Go, J. Won, D. F. Kune, E. Jeong, Y. Kim, and K. Park. Gaining Control of Cellular Traffic Accounting by Spurious TCP Retransmission. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[10] M. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park. Kargus: a Highly-scalable Software-based Intrusion Detection System. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[11] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: A Highly scalable user-level TCP stack for multicore systems. In *Proceedings of the USENIX conference on Networked systems design and implementation (NSDI)*, 2014.

[12] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *Proceedings of the USENIX conference on Networked systems design and implementation (NSDI)*, 2012.

[13] S. Woo, E. Jeong, S. Park, J. Lee, S. Ihm, and K. Park. Comparison of caching strategies in modern cellular backhaul networks. In *Proceeding of the annual international conference on Mobile systems, applications, and services (MobiSys)*, 2013.