CliMB: Enabling Network Function Composition with Click Middleboxes

Rafael Laufer, Massimo Gallo
Nokia Bell Labs
first.last@nokia.com

Diego Perino*
Telefonica Research
diego.perino@telefonica.com

Anandatirtha Nandugudi
INRIA
anadatirtha.nandugudi@inria.fr

ABSTRACT

Click has significant advantages for middlebox development, including modularity, extensibility, and reprogrammability. Despite these features, Click still has no native TCP support and only uses nonblocking I/O, preventing its applicability to middleboxes that require access to application data and blocking I/O. In this paper, we attempt to bridge this gap by introducing Click middleboxes (CliMB). CliMB provides a full-fledged modular TCP layer supporting TCP options, congestion control, both blocking and nonblocking I/O, as well as socket and zero-copy APIs to applications. As a result, any TCP network function may now be realized in Click using a modular L2-L7 design. As proof of concept, we develop a zero-copy SOCKS proxy using CliMB that shows up to 4x gains compared to an equivalent implementation using the Linux in-kernel network stack.

CCS Concepts

•**Networks** → **Middle boxes / network appliances; Programmable networks;**

Keywords

Click router, Middle boxes, TCP

1. INTRODUCTION

Middleboxes are responsible for a variety of network functions, such as network address translation, traffic filtering, and load balancing, and are now as prevalent in enterprise networks as routers and switches [18]. Due to performance requirements, middleboxes have initially been implemented in hardware and provisioned for peak load, being thus quite expensive and providing limited flexibility. More recently, software middleboxes have gained popularity because of their increased flexibility and ability to scale with network load. This trend towards network function virtualization (NFV) ignited novel approaches for deploying software middleboxes, ranging from virtual machines [7, 13] to innovative software designs [2, 14]. Nonetheless, the step towards a practical architecture enabling modular L2–L7 network functions has not yet been made. This would promote code reuse and cross-layer optimizations (e.g., zero copy) in middleboxes, as well as allow the rapid deployment of new services.

Among existing approaches, Click [11] is likely the top candidate for such an architecture due to its modularity and extensibility. In particular, Click uses fine-grained elements that connect together to realize the desired network function. This allows significant code reuse, as the same element is used for different network functions. Click elements can also be reconfigured on-the-fly and even the entire router functionality may be remotely replaced via hot swapping, without traffic disruption. Support for in-kernel execution and multithreading are also available for high performance.

In spite of these advantages, two key limitations may still prevent Click from being an integrated middlebox platform. First, Click does not offer native TCP support for network applications, restricting potential cross-layer optimizations and stack customization. Instead, applications written in Click must resort to the OS stack, which lacks flexibility and may lead to severe I/O bottlenecks [8, 16]. Second, Click has no support for blocking I/O primitives. This, in turn, may impose a burden on developers to build applications with asynchronous nonblocking I/O or waste CPU resources with busy-waiting.

To overcome these limitations, we introduce in this paper an architecture called CliMB that enables network function composition in Click. Along with the original Click elements, CliMB enables users to overhaul the entire network stack, if desired. For fast packet I/O, CliMB uses Intel® DPDK [8] to achieve line rate in user space. For applications, CliMB introduces a standards-compliant modular TCP implementation supporting TCP options, congestion control, and RTT estimation. It exposes a socket API for easily porting existing applications and a zero-copy API for high performance. In addition to allowing nonblocking I/O, CliMB introduces blocking I/O support in Click. In this case, the CPU context is saved if an I/O request cannot be promptly completed and restored once it completes. This provides network applications with the illusion of running uninterrupted, eliminating complexity in their design.

With these features, CliMB can be used to deploy a vast range of network functions and services requiring cross-layer optimizations, such as L7 congestion control and zero copy, or TCP connection termination, particularly useful for Split

*Work done at Nokia Bell Labs.

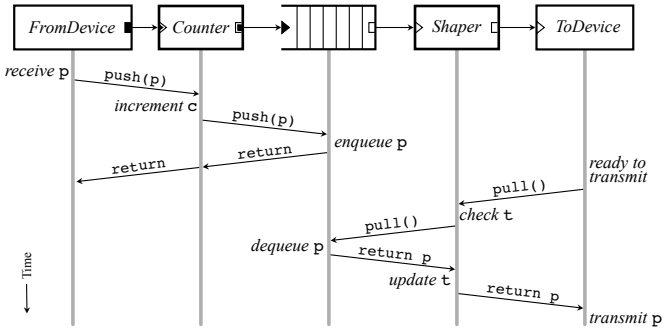

Figure 1: Push and pull control flow in Click. Upon arrival, a packet is pushed to downstream elements until it is either stored, dropped, or transmitted. When the output interface is ready, a packet is pulled from upstream elements, if one is available.

TCP, L7 load balancing, L7 firewall, TLS/SSL termination proxy, HTTP caching proxy, and reverse proxy. As proof of concept, we implement a zero-copy SOCKS proxy server to show how CliMB can be used to realize modular full-stack middleboxes currently unsupported in Click. The server uses synchronous I/O multiplexing primitives provided by CliMB to efficiently handle I/O events. We compare its performance with an equivalent Linux implementation and show up to 4x gains in throughput.

The remainder of the paper is organized as follows. In Section 2, we provide an overview of Click and briefly discuss its limitations to develop high-performance TCP middleboxes. Section 3 then describes the proposed CliMB architecture and its key components, namely, high-speed packet I/O, blocking I/O, and Click TCP. In Section 4, we present our preliminary results and show that, in addition to enabling L2–L7 modularity, CliMB is also able to significantly improve performance. Section 5 reviews the related work and explains how CliMB is different than previous approaches. Finally, Section 6 concludes the paper.

2. CLICK MODULAR ROUTER

2.1 Overview

Click is a software architecture for building modular and extensible routers. A router in Click is built from a set of fine-grained packet processing modules called *elements*, which implement simple functions (e.g., IP route lookup). A router *configuration* connects these elements together into a directed graph, and packets traverse the edges of this graph. Depending on the selected elements in the graph and the connections among them, different network functions can be implemented (e.g., IP router, Ethernet switch).

Connections between a pair of elements are established by *ports*. Each element may define any number of input and output ports to connect to other elements. These ports must operate in either *push* or *pull* mode, depending on the element implementation. On a push connection, the packet starts at the source element and moves to the destination element downstream. On a pull connection, in contrast, the destination element requests a packet to the upstream source element, which returns a packet if one is available or a null pointer otherwise. In addition to push or pull, a port may also be *agnostic* and behave as either push or pull depending on the other port it is connected to.

Figure 1 shows these concepts using a simple router that forwards packets from one interface to another, measuring the number of received packets and shaping output traffic. For this purpose, *Counter* has a local counter `c`, *Queue* has a packet list, and *Shaper* has a local timestamp `t` storing the last time that a packet was pulled, used to limit the pulling rate. In the figure, input ports are depicted as triangles and output ports as rectangles. Push ports are black and pull ports are white; agnostic ports are shown as either push or pull with a double outline.

In its underlying implementation, Click employs a task queue and runs an infinite loop that executes each task in sequence. Tasks are basically element-defined functions that require frequent CPU scheduling, and initiate a sequence of push or pull requests. All packet processing in Click is initiated by tasks. In the previous example, both *FromDevice* and *ToDevice* have periodic tasks interacting with network interfaces to receive or transmit packets, respectively. Most elements, however, do not require their own task for CPU access, since their `push` and `pull` methods are implicitly scheduled when called by another task. This is notably the case of the three intermediate elements in Figure 1. Once a packet is pushed or pulled through the router graph, it continues to be processed at each element along a path until it is stored, dropped, or transmitted.

2.2 Limitations

Click has significant advantages as a platform for the design and implementation of middleboxes, including modularity, remote reprogrammability, multithreading, and both user and kernel space implementations. Despite these features, a couple of limitations may still prevent Click from being a complete solution for middleboxes:

No transport layer: Click does not have a native TCP layer, which prevents the deployment of several middleboxes requiring access to L7 data. Sockets (using the OS stack) are in fact available in user space, but require a separate element for each TCP connection, making it impractical for real-world applications with dynamic connections. Inefficiencies in the OS stack may also prevent elements from achieving line rate [8].

Nonblocking I/O: Once a task is scheduled in Click, it runs to completion. There is no way, for instance, to block a task until a set of I/O operations complete. As a result, even if a transport layer did exist in Click, applications would have to employ nonblocking I/O to transfer network data, imposing a burden on developers or wasting CPU resources with busy-waiting. Moreover, porting applications relying on blocking primitives to Click would be impractical.

3. CLIMB ARCHITECTURE

To overcome these limitations, we introduce an architecture called Click middleboxes (CliMB) that enables novel network functions using a modular and customizable L2–L7 stack, blocking I/O, and high-speed packet I/O. This section provides a brief overview of CliMB and its components. Sections 3.1 and 3.2 describe CliMB packet I/O and blocking I/O design, respectively. Section 3.3 explains the TCP implementation in CliMB, and Section 3.4 reviews its socket and zero-copy APIs.

3.1 Packet I/O

CliMB currently runs as a single thread in user space Click. To provide high-speed packet I/O without the kernel network stack, we implement an element that uses Intel® Data Plane Development Kit (DPDK) to directly interface with 10 GbE cards. The element has a task that frequently polls the network card to fetch received packet batches. Each fetched packet is then wrapped in a Click packet data structure and pushed out or temporarily buffered for future pull calls.

The DPDK functionality in CliMB is similar to the one provided by FastClick [3], with key improvements. In addition to the push mode, our DPDK element also supports pull mode for transmissions and receptions. Notification signals can also be received or transmitted, indicating queue occupancy from or to other elements. Additionally, we exploit the NIC to perform both IP and TCP checksum offloading as well as hardware flow control to avoid buffer overflows.

Using a single core, our DPDK element is able to achieve 10 Gbps with 64-byte packets in both push and pull modes, during both transmission and reception.

3.2 Blocking I/O

Click natively supports nonblocking I/O through its `push` and `pull` methods; however, it has no support for blocking I/O. For network applications, blocking I/O has the advantage of being simpler to implement on and often providing equivalent performance. Moreover, blocking I/O is also required to support efficient system calls (e.g., `poll`) for socket I/O multiplexing.

To provide middlebox applications with a broader range of I/O options, CliMB implements blocking I/O in Click. We introduce the concept of *blocking tasks*, which can yield the CPU to other eligible tasks if a given condition does not hold, e.g., an I/O request that cannot be promptly completed. When rescheduled, the task resumes exactly where it left off, providing users with the illusion of continuity. This mimics the behavior of user-level threads in Linux under I/O requests.

Task scheduling is still cooperative, as each task must explicitly yield the CPU if waiting for an I/O request to complete. This does not occur in the task itself, but rather within the API-provided functions (Section 3.4) when an I/O request cannot be immediately completed. When the I/O request is finally completed, the task is rescheduled for execution. Blocking tasks are backward compatible with regular tasks, requiring no modifications to the Click task scheduler.

Each blocking task has its own call stack and runs as a separate context within the same user-level process. Context switching between tasks is light-weight, saving and restoring a handful of registers required for task execution. CliMB uses low-level functions to save and restore the user-level context, just as in POSIX threads. Different from threads, however, CliMB has complete control over the task scheduler and relies on cooperative, as opposed to preemptive, scheduling.

3.3 Click TCP

To enable modular L2–L7 middlebox development, CliMB implements a TCP transport layer in Click. Our Click TCP (cTCP) implementation is in full compliance with standards (RFCs 793 and 1122), supporting TCP options (RFC 7323), NewReno congestion control (RFCs 5681 and 6582), as well as timer management and RTT estimation (RFC 6298). Overall, cTCP is a compound element composed of more than 40 elements that jointly implement the TCP protocol. These elements can be easily extended or replaced to experiment with alternatives (e.g., congestion control).

Figure 2 depicts the cTCP configuration graph for incoming TCP packets. In essence, elements access and/or modify the TCP control block (TCB) of the flow as the packet moves along its path. The vertical paths represent the direction that the received packets usually take. Other paths represent a disruption in the expected packet flow, e.g., *TCPCheckSeqNo* sends an ACK if all data in the packet is out of the receive window.

Most elements in cTCP process packets using only the TCB information. For example, *TCPTrimPacket* trims off any out-of-window data from the packet, ensuring that the remaining data is within the receive window. *TCPReordering* enforces in-order delivery by buffering out-of-order packets and pushing them out in sequence once the gap is finally filled. *TCPProcessRst*, *TCPProcessSyn*, *TCPProcessAck*, and *TCPProcessFin* inspect the respective TCP flags and act according to the standards. In presence of new data, *TCPProcessTxt* clones the packet, strips its headers, and places it on the RX queue for the application to process.

Other elements in cTCP require information previously computed by another upstream element. This is supported in Click through *packet annotations*, which are specific metadata carried within the packet. Packet annotations used in cTCP include:

TCB pointer: The TCB table is stored in *TCPInfo* and accessed by other elements using static functions. For each packet, *TCPFlowLookup* looks the TCP flow up in the table and sets a pointer in the annotations to allow downstream elements to access/modify the TCB.

RTT measurement: *TCPAckOptionsParse* computes the round-trip time (RTT) from the TCP timestamp option and sets it as an annotation. If this option is not supported, *TCPRtxEnqueue* timestamps each packet before storing it in the retransmission queue. In both cases, *TCPEstimateRTT* uses these annotations to estimate the RTT and update the retransmission timeout.

Acknowledged bytes: *TCPProcessAck* computes the number of acknowledged bytes in each packet and sets an annotation used by *TCPNewRenoAck* to increase the congestion window. If this number is zero and a few other conditions hold, the packet is considered a duplicate ACK, which may trigger a fast retransmission.

Flags: A few flags are used to indicate certain actions to other elements, such as sending an acknowledgment back after processing the packet. This flag is set, for instance, by *TCPProcessTxt* when the received packet has new data. The flag is checked by *TCPAckRequired* and the packet is pushed out if the flag is set, and dropped otherwise.

Finally, timers are used in cTCP for retransmission timeouts, delayed ACKs, and TCP keepalive messages. A retransmission timeout occurs when transmitted data is not acknowledged after some time. Acknowledgments are sent once for every two received data packets or if 500 ms pass since the last data packet is received. TCP keepalive messages are periodically sent to test for connectivity. Due to

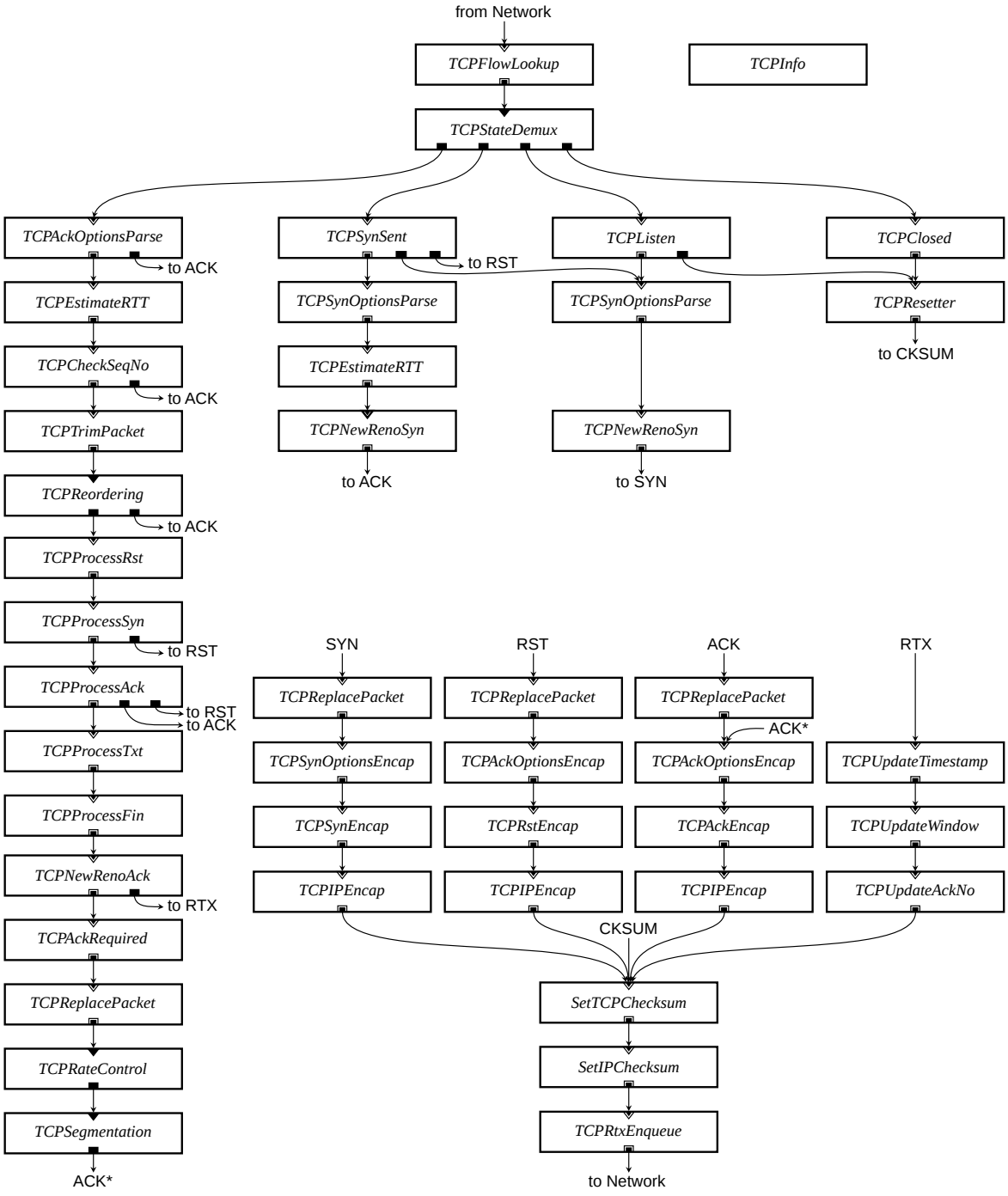

Figure 2: Click TCP (cTCP) configuration graph for incoming network packets.

space constraints, the cTCP configuration graphs for timers and APIs are omitted.

3.4 Application Programming Interfaces

In CliMB, one or multiple Click elements implement the application logic and communicate with cTCP using APIs. The design of the cTCP APIs is driven by two main goals: (i) minimize the effort required to port existing applications to cTCP; and (ii) provide users with zero-copy primitives to guarantee high performance. As these are conflicting objectives, cTCP provides two APIs from which application developers may choose to meet their requirements.

Socket API: Application elements interface with the cTCP layer using the well-known socket API. For each socket function (e.g., `send`), cTCP provides a corresponding function (e.g., `click_send`) that works both in blocking or nonblocking mode. As in Linux, the operation mode is set on a per-socket basis via the `SOCK_NONBLOCK` flag. Applications in CliMB run within a blocking task, which is unscheduled when an I/O request cannot be promptly completed.

This occurs, for instance, when the RX queue is empty and `click_recv` is called. Once the I/O request completes, the task is rescheduled for execution again.

In addition to blocking, sockets may also operate in non-blocking mode. In this case, the aforementioned calls would return right away with an error. To avoid busy-waiting on I/O, a `click_poll` function is provided to monitor multiple socket descriptors at the same time. If no socket is ready to perform I/O, the function blocks the calling task. In future work, cTCP will also provide functions based on `select` and `epoll_wait`.

The socket API is ideal for porting existing applications to CliMB by just replacing the socket system calls. However, memory copies are still required to transfer data between the application and the cTCP layer.

Zero-copy API: cTCP provides an alternative to avoid data copy during transmissions and receptions. For transmissions, applications must first allocate a packet directly from the DPDK memory pool and write data into it, leaving some headroom for protocol headers. The application then sends the data packet with the zero-copy `click_push` function. For receptions, it calls `click_pull` to retrieve a packet; these are the actual packets where the data came in. To amortize per-packet overhead, both functions also send and receive batches, and may operate in either blocking or nonblocking mode. Using the zero-copy API, it is possible, for instance, to build efficient applications that never have to copy data around, as shown in the next section.

4. PRELIMINARY RESULTS

We perform two experiments to validate the cTCP implementation. First, we run a microbenchmark where a file is transferred from a client to a server running on two machines directly connected to each other. The goal is to determine if cTCP is able to sustain line rate. We then use the socket and zero-copy APIs to implement a SOCKS4 proxy server through which multiple clients and servers connect. Our experimental setup consists of 5 machines using Intel Xeon® 16-core E5-2630v3 2.4GHz processors, 16 GB RAM, and equipped with an Intel® 82599ES network card containing two 10 GbE interfaces. The machines run Ubuntu 14.04 (kernel 4.4.0-14), Click 2.1, and Intel® DPDK 2.2.0.

File transfer: To first validate the cTCP implementation in CliMB, we ran a simple test where a cTCP client application connects to a cTCP server, sends a 10 GB file using the zero-copy API (*i.e.,* `click_push`) and, when finished, gracefully closes the connection. The cTCP server listens for incoming connections and, once the three-way handshake is over, waits for incoming data using the zero-copy API (*i.e.,* `click_pull`). When a data packet is received, the server just discards it and waits for more data. Both the client and the server tasks use blocking I/O to wait if the TX queue is full or if the RX queue is empty, respectively. Using a single core at each host, our cTCP implementation is able to achieve the maximum data throughput of 9.41 Gbps.

SOCKS4 proxy server: As a proof of concept, we built a zero-copy proxy server that acts on behalf of clients. The server consists of a single element running a blocking task that implements the Socket Secure 4 (SOCKS4) protocol. After it first starts listening for incoming connections, the proxy runs a loop that calls `click_poll` to monitor all open sockets and handle any pending I/O. Once a connection is accepted, it waits for the client to send the SOCKS request containing the IP address and TCP port of the destination server. With this information, a connection is then established to the server and a response is sent back to notify the client that its request was granted. From this moment on, any data received from either the client or the server is forwarded to the other party using the zero-copy API.

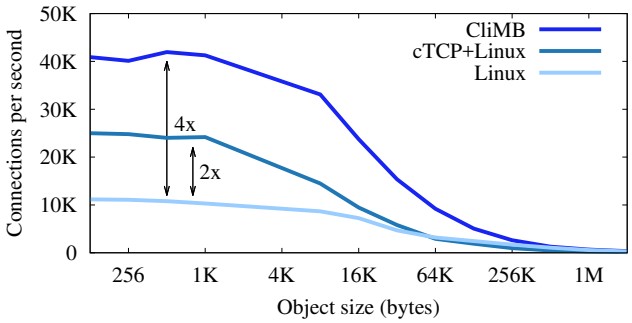

Figure 3: Performance of a SOCKS4 proxy server in both CliMB and Linux for different object sizes.

For performance evaluation, we compare the proposed CliMB SOCKS4 proxy server with its counterpart without DPDK (i.e., using the *FromDevice* and *ToDevice* elements) and with an equivalent implementation entirely in Linux. We use two machines as clients (`curl`), one as the proxy server, and two as web servers (`lighttpd`). Each client machine runs 32 `curl` processes that continuously open a connection to a web server via the SOCKS proxy, request some data, and then close the connection; this experiment lasts for 60 seconds. Each server machine runs `lighttpd` with 15 threads to accommodate the network load. Figure 3 shows the number of connections per second for various object sizes requested by the clients. We see that CliMB outperforms Linux by up to 4x for small objects. When using Linux for packet I/O, cTCP still has a 2x gain; the remaining gain in CliMB is thus due to DPDK acceleration.

5. RELATED WORK

Click and its extensions: Click and its modular data plane are initially proposed in [11] and, since then, have been extended in multiple directions. For instance, high-speed packet I/O is now available in Click [3, 8, 16]. To increase throughput, support for Click elements running on GPUs has also been recently added [10, 19]. In addition, the increasing popularity of NFV motivated the design of a lightweight OS based on Click [13]. Although reaching line rate and enabling simple middlebox functionality, these systems do not have a TCP layer. In contrast, CliMB extends Click to support a full-fledged TCP implementation, which enables modular network function composition for complex middleboxes.

Modular NFV: Click has inspired other modular network function management systems, such as Slick [2], Openbox [6], and Elastic Edge [14]. These systems focus on control plane operations, such as data plane element placement, network function scaling, and traffic steering. For the data plane, FlowOS [5] is proposed as a middlebox platform that enables flow processing, but without TCP support. CoMb [17] and xOMB [1] use Click to consolidate middleboxes through

the composition of different L7 elements. Both rely on the OS implementation for packet I/O and transport layer, reducing customization and performance.

Frameworks enabling stack customization of L2–L7 are proposed in [15, 20]. PDP [15] introduces the design of a programmable data plane focusing on hardware offloading. In [20] a preliminary design of a modular middlebox platform based on mTCP [9] is presented. In contrast to existing work, CliMB introduces cTCP and blocking I/O primitives in Click to exploit its built-in modularity and extensibility, while also offering easy portability for existing applications, L2–L7 stack customization, and cross-layer optimization.

User-level stack: Efficient user-level network stacks have been proposed to overcome I/O inefficiencies of operating systems [4, 9, 12]. IX [4] is a data plane operating system that separates the control plane from the data plane. mTCP [9] is a user-level TCP implementation proposed for multicore systems. Sandstorm [12] proposes a clean-slate network stack providing zero-copy APIs for each protocol layer. Different from [4, 9, 12], CliMB is based on the Click architecture to guarantee modularity and extensibility to middleboxes. It provides a user-space TCP implementation offering both a L2–L7 zero-copy API for high performance as well as a socket API for ease of portability.

6. CONCLUSIONS

In this paper we presented CliMB, an architecture that enables the design and implementation of Click middleboxes. CliMB introduces the concept of blocking tasks in Click to allow network applications to efficiently wait on I/O without consuming resources. It also provides a full-fledged modular TCP implementation, supporting blocking and nonblocking I/O as well as socket and zero-copy APIs for application portability and high performance. With the introduction of a TCP layer, CliMB enables L2–L7 stack customization and cross-layer optimizations that were not possible before in Click. As proof of concept, we implemented a SOCKS4 proxy server in CliMB whose preliminary results provide encouraging performance gains. CliMB source code will be publicly available in the near future.

Our future work will focus first in extending CliMB for multicore scalability, enabling flow-level core affinity from L2 to L7. This will enable TCP middleboxes to achieve high speeds without requiring synchronization primitives. With high performance in multicore architectures, our goal is to explore Click reprogrammability for redesigning the middlebox functionality on-the-fly. This will enable NFV provisioning by allowing controllers to spawn new modular L2–L7 services without incurring the high I/O overhead of virtualization. Finally, our intention is to provide a NFV control plane to enable network function orchestration across the network, and balance the load among different middleboxes according to their resource usage.

7. REFERENCES

[1] J. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xOMB: Extensible Open Middleboxes with Commodity Servers. In *Proc. of ACM/IEEE ANCS*, 2012.

[2] B. Anwer, T. Benson, N. Feamster, and D. Levin. Programming Slick Network Functions. In *Proc. of ACM SOSR*, 2015.

[3] T. Barbette, C. Soldani, and L. Mathy. Fast Userspace Packet Processing. In *Proc. of ACM/IEEE ANCS*, 2015.

[4] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proc. of USENIX OSDI*, 2014.

[5] M. Bezahaf, A. Alim, and L. Mathy. FlowOS: A Flow-based Platform for Middleboxes. In *Proc. of ACM HotMiddleboxes*, 2013.

[6] A. Bremler-Barr, Y. Harchol, and D. Hay. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *Proc. of ACM SIGCOMM*, 2016.

[7] J. Hwang, K. K. Ramakrishnan, and T. Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *Proc. of USENIX NSDI*, 2014.

[8] Intel DPDK framework. http://dpdk.org.

[9] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proc. of USENIX NSDI*, 2014.

[10] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, and S. Moon. NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors. In *Proc. of EuroSys*, 2015.

[11] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 2000.

[12] I. Marinos, R. N. Watson, and M. Handley. Network Stack Specialization for Performance. In *Proc. of ACM SIGCOMM*, 2014.

[13] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *Proc. of USENIX NSDI*, 2014.

[14] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A Framework for NFV Applications. In *Proc. of ACM SOSP*, 2015.

[15] D. Perino, M. Gallo, R. Laufer, Z. B. Houidi, and F. Pianese. A Programmable Data Plane for Heterogeneous NFV Platforms. In *Proc. of IEEE INFOCOM SWFAN workshop*, 2016.

[16] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *Proc. of USENIX Security*, 2012.

[17] V. Sekar, N. Egi, S. Ratnasamy, M. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *Proc. of USENIX NSDI*, 2012.

[18] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone Else's Problem: Network Processing As a Cloud Service. In *Proc. of ACM SIGCOMM*, 2012.

[19] W. Sun and R. Ricci. Fast and Flexible: Parallel Packet Processing with GPUs and Click. In *Proc. of IEEE/ACM ANCS*, 2013.

[20] S. Woo, K. Jang, D. Han, and K. Park. Towards an Open Middlebox Platform for Modular Function Composition. In *Proc. of USENIX NSDI, poster*, 2012.