ClickNF: a Modular Stack for Custom Network Functions

Massimo Gallo and Rafael Laufer, *Nokia Bell Labs*

https://www.usenix.org/conference/atc18/presentation/gallo

This paper is included in the Proceedings of the
2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

ClickNF: a Modular Stack for Custom Network Functions

Massimo Gallo and Rafael Laufer
Nokia Bell Labs

Abstract

Network function virtualization has recently allowed specialized equipment to be replaced with equivalent software implementation. The Click router was a first step in this direction, defining a modular platform for generalized packet processing. Despite its major impact, however, Click does not provides native L4 implementation and only uses nonblocking I/O, limiting its scope to L2-L3 network functions. To overcome these limitations we introduce ClickNF, which provides modular transport and application-layer building blocks for the development of middleboxes and server-side network functions. We evaluate ClickNF to highlight its state-of-the-art performance and showcase its modularity by composing complex functions from simple elements. ClickNF is open source and publicly available.

1 Introduction

Software-defined networking had a significant impact on the packet forwarding infrastructure, providing flexibility and controllability to network and datacenter operators [37]. In a similar trend, network function virtualization (NFV) is sparking novel approaches for deploying flexible network functions [19], ranging from virtual machine orchestration [24, 36, 34] to new packet processing frameworks [8, 40]. Network functions can combine packet forwarding and simple header rewriting with awareness of stateful transport logic, and possibly execute complex application-layer operations.

A modular L2–L7 data plane would offer several advantages for the development of new network functions, such as decoupling state and packet processing, extensibility of fine-grained protocol behavior, module reuse, and a simplification of cross-layer protocol optimizations and debugging. Among existing approaches, Click [29] is arguably the best starting point for such an architecture due to its modularity and extensibility. However, several functionalities are still missing to make Click into a full-stack modular data plane for network functions. First, it lacks L4 native implementation, preventing cross-layer optimizations and stack customization. Second, it has no support for blocking I/O primitives, forcing developers to use more complex asynchronous non-blocking I/O. Third, Click applications must resort to the OS stack, which leads to severe I/O bottlenecks. Finally, despite recent improvements, Click does not support hardware offloading and efficient timer management preventing it to scale at high-speed in particular scenarios.

In this paper we introduce ClickNF, a framework that overcomes the aforementioned limitations and enables L2–L7 modular network function development in Click. Along with legacy Click elements, ClickNF enables developers to overhaul the entire network stack, if desired. First, it introduces a modular TCP implementation that supports options, congestion control, and RTT estimation. Second, it introduces blocking I/O support, providing applications with the illusion of running uninterrupted. Third, it exposes standard socket, zero-copy, and socket multiplexing APIs as well as basic application layer building blocks. Finally, to improve scalability, ClickNF integrates I/O acceleration techniques first introduced in Fastclick [9], such as Data Plane Development Kit (DPDK) [33] and batch processing with additional support for hardware acceleration, as well as an improved timer management system for Click.

ClickNF can be used to deploy a vast class of network functions. For middleboxes, TCP termination is needed for Split TCP, L7 firewalls, TLS/SSL proxies, HTTP caches, etc. At the network edge, ClickNF can be used to implement high-speed modular L7 servers using socket multiplexing primitives to handle I/O events efficiently. As proof of concept, we compose an HTTP cache server with optional SSL/TLS termination and a SOCKS4 proxy. We show that ClickNF provides equivalent performance and scalability as existing user-space stacks while enabling L2–L7 modularity.

The paper is organized as follows. Section 2 describes ClickNF design. Section 3 details our TCP implementation in Click and Section 4 presents application layer modularity. Section 5 highlights a number of original aspects about the ClickNF implementation that are evaluated in Section 6. Section 7 reviews the related work and Section 8 concludes the paper.

2 ClickNF

ClickNF leverages Click [29], a software architecture for building modular and extensible routers. Before introducing its design, we first provide an overview of Click.

2.1 Background

A router in Click is built from a set of fine-grained packet processing modules, *elements*, implementing simple functions (e.g., IP routing). A *configuration* file connects these elements together into a directed graph, whose edges specify the path that packets shall traverse. Depending on the configuration, users can implement network functions of arbitrary complexity (e.g., switch).

Each element may define any number of input and output *ports* to connect to other elements. Ports operate in either *push* or *pull* mode. 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 from the upstream one, which returns a packet if 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 port it is connected to.

In its underlying implementation, Click employs a task queue and a timer priority queue. An infinite loop runs tasks in sequence and timers at expiration. Tasks are element-defined functions that require CPU scheduling, and initiate a sequence of push or pull requests. Most elements, however, do not require their own task, since their `push` and `pull` methods are called by a scheduled task. Timer callback functions are similar to tasks, except for being scheduled at a particular time.

2.2 Design

Network protocol stacks are typically implemented as monolithic packages, either in kernel or user-space. Network function developers often experience hurdles when attempting to debug and customize their software to obtain the desired effects, as the inner workings of the stacks are not exposed. Indeed, recent work [22, 17, 39, 38] advocates that legacy network stacks prevent innovation due to the lack of flexibility and propose to move some of their functionalities outside of the data path.

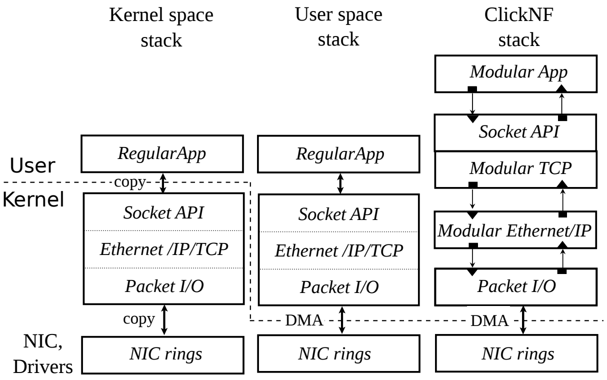


Figure 1: ClickNF design compared to alternatives.

Modular, configurable, and extensible transport protocols were proposed in the past by the research community [16, 13] and by the Linux kernel one [1] constituting a first step in the right direction. Our goal, is along the same lines but broader. ClickNF aims to give developers unfettered access to the entire stack by providing a framework for the construction of modular L2-L7 network functions without the concerns for the correctness of its behavior nor the constraints added by event-driven domain-specific APIs [26].

The design of ClickNF combines the modularity and flexibility of Click with high-speed packet I/O and ready-made protocol building blocks for transport and application-layer features. Figure 1 compares ClickNF design against legacy OSs and user space stacks. In contrast with other approaches that conceal network stack complexity into a monolithic package or does not introduce modularity at all layers, we decompose the full L2–L7 stack into several simple elements that can be individually replaced, modified or removed by rewiring the configuration file, providing a level of flexibility that is not available with alternative solutions. Additionally, elements can be aggregated into a single macro-element to hide complexity when desired. The rationale behind this fine-grained decomposition is twofold. First, simple elements allow the modification and control of each aspect and mechanism of network protocols. This enables module reuse in other contexts, such as recycling existing congestion control strategies to implement new protocols like QUIC [30], or new strategies such as BBR [14] or DCTCP [6] with little effort. Second, this approach helps decoupling protocol state management and packet processing, simplifying complicated tasks such as full state migration between servers or across heterogeneous hardware (e.g., between CPUs and smart NICs).

In the rest of the paper we focus on the description and evaluation of ClickNF transport and application layers, and on some important implementation details that allows ClickNF to sustain line-rate.

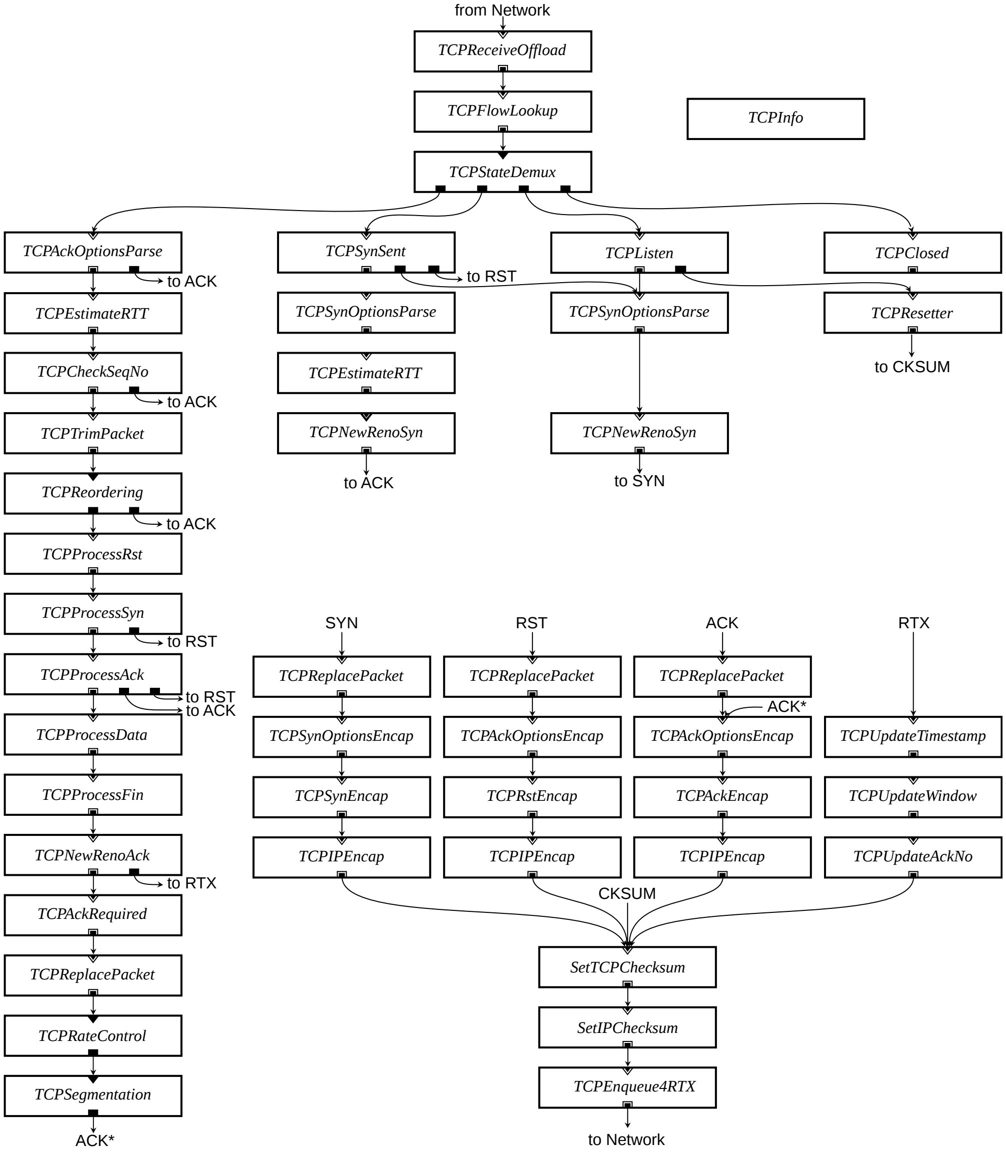


Figure 2: cTCP configuration for incoming network packets.

3 Click TCP

Our modular Click TCP (cTCP) implementation is compliant with IETF standards (RFCs 793 and 1122) and supports TCP options (RFC 7323), New Reno congestion control (RFCs 5681 and 6582), timer management, and RTT estimation (RFC 6298). In this section we describe the cTCP element graphs used to process incoming and outgoing TCP packets.

3.1 Incoming packets

The key element of cTCP is *TCPInfo*, which enables state decoupling by providing other elements with access to important data structures (*i.e.,* TCP Control Block). Figure 2 shows the cTCP element graph for processing incoming packets. In essence, elements access and/or modify the TCP control block (TCB) via the *TCPInfo*, as the packet moves along the edges of the graph. The vertical paths are the directions that most received packets take. The long element sequence on the left represents the packet processing of an established TCP connection. The three other paths to the right take care of special situations, such as connection establishment and termination. Other paths in the graph represent a disruption in the expected packet flow, e.g., *TCPCheckSeqNo* sends an ACK back if the data is outside the receive window.

Most elements in cTCP only require the TCB to process packets. For instance, *TCPTrimPacket* trims off any out-of-window data from the packet. *TCPReordering* enforces in-order delivery by buffering out-of-order packets and releasing them in sequence once the gap is filled.

Elements like *TCPProcess*{*Rst, Syn, Ack, Fin*} inspect the respective TCP flags and react accordingly. In presence of new data, *TCPProcessData* clones the packet (*i.e.,* only packet's metadata are copied) and places it on the RX queue for the application. After receiving three duplicate ACKs, *TCPNewRenoAck* retransmits the first unacknowledged packet. Related elements, such as *TCPNewRenoSyn* and *TCPNewRenoRTX*, handle initialization and retransmissions in congestion control.

In addition to the TCB, other cTCP elements require information previously computed by other elements. This is supported in Click via *packet annotations*, *i.e.,* packet metadata. cTCP packet annotations include:

TCB pointer: The TCB table is stored as a per-core hash table in *TCPInfo* and accessed by other elements using static functions. For each packet, *TCPFlowLookup* looks the TCP flow up in the table and sets the TCB annotation to allow other elements to easily access/modify the TCB avoiding multiple flow table lookups.

RTT measurement: *TCPAckOptionsParse* computes the RTT from the TCP timestamp, Karn's algorithm, and sets it as an annotation. If TCP timestamps are not provided by NICs or by packet I/O elements, *TCPEnqueue4RTX* timestamps each transmitted packet before storing it in the retransmission (RTX) 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 it as an annotation. This is later read by *TCPNewRenoAck* to increase the congestion window. If this number is zero and a few other conditions hold (e.g., the receive window is unchanged), the packet is considered a duplicate ACK and may trigger a fast retransmission.

Flags: TCP flags are used to indicate certain actions to other elements. For instance, *TCPProcessData* sets a flag when the received packet has new data. *TCPAckRequired* then checks this flag and, if set, pushes the packet out to trigger an ACK transmission.

3.2 Outgoing packets

Figure 3 shows the cTCP element graph for processing outgoing packets. Applications send data using socket or zero-copy APIs (Section 3.3) that invoke static functions in *TCPSocket*. For each socket call, this element looks up the socket file descriptor in a per-core socket table in *TCPInfo*. For transmissions, *TCPNagle* first checks if the packet should be sent according to Nagle's algorithm. *TCPRateControl* then verifies whether send and congestion windows allow packet transmission. If so, *TCPSegmentation* breaks the data up into MTU-sized packets, and *TCPAckOptionEncap*, *TCPAckEncap*, and *TCPIPEncap* prepend TCP and IP header respectively. Before sending it to lower layers, *TCPEnqueue4RTX* clones the packet and saves it in the retransmission queue until acknowledged by the other end. To initiate a TCP connection, *TCPSynOptionEncap* and *TCPSynEncap* generate the TCP options and header and forward the packet downstream. Similarly, to terminate the connection, *TCPAckOptionEncap* and *TCPFinEncap* form a TCP packet with the FIN flag set.

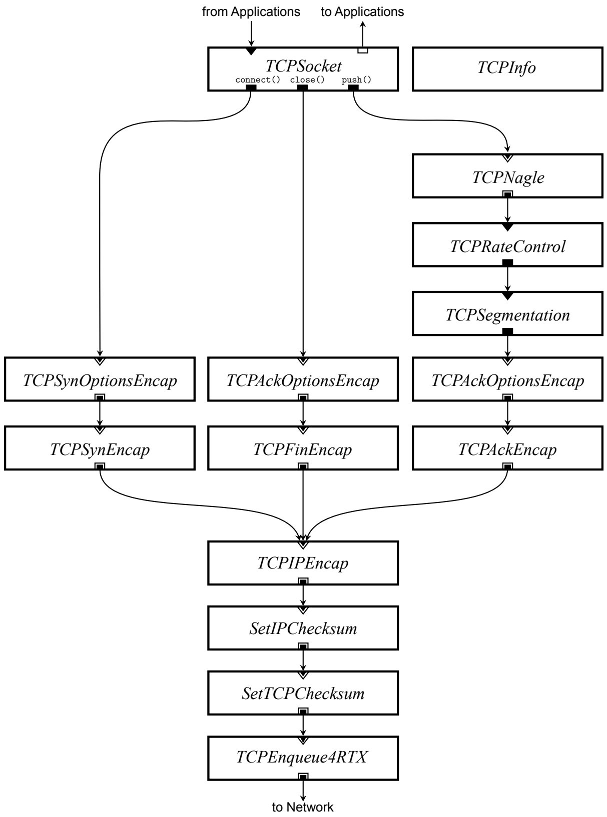


Figure 3: cTCP configuration for outgoing packets.

3.3 Transport APIs

cTCP APIs are designed with two contrasting objectives in mind: (*i*) minimize the efforts required to port application logic in ClickNF; and (*ii*) provide primitives to guarantee high performance at the cost of more complex development. We therefore provide two APIs to interact with the ClickNF transport layer, and one for socket I/O multiplexing.

Socket API: For each socket system call (e.g., `send`), cTCP provides a corresponding function (e.g., `click_send`) for both blocking or non-blocking mode. As in Linux, the operation mode is set on a per-socket basis using the `SOCK_NONBLOCK` flag. In case of blocking sockets, the application is blocked when waiting on I/O; in case of non-blocking sockets, the socket calls return immediately.

Zero-copy interface: In addition to standard Socket API, cTCP provides `click_push` and `click_pull` functions to enable zero-copy operations. For transmissions, applications first allocate a packet and write data to it before pushing it down. cTCP then adds the protocol headers and transmits the packet(s) to the NIC. For receptions, packets are accessed, processed, and placed into the RX queue of applications. To amortize per-packet overhead, both functions can also send and receive batches, and operate in either blocking or non-blocking mode.

Socket I/O multiplexing: To avoid busy-waiting on I/O, cTCP provides a `click_poll` and a `click_epoll_wait` functions to multiplex events from several socket file descriptors. As in the regular epoll API provided by Linux kernel, applications must first register the monitored socket file descriptors with `click_epoll_ctl` and then use `click_epoll_wait` to wait on I/O.

3.4 Timer Management

In Click, timers corresponds to tasks scheduled at a given time in the future. TCP timers are used for retransmissions, keepalive messages, and delayed ACKs. Their implementation in cTCP is critical for performance and scalability reasons. Figure 4 shows cTCP timers' configuration. After a retransmission timeout task is scheduled, *TCPTimer* dequeues the head-of-line packet from the RTX queue and pushes it out. *TCPUpdateTimestamp*, *TCPUpdateWindow*, and *TCPUpdateAckNo* update the respective TCP header fields. Similarly when a keepalive timeout expires, *TCPTimer* pushes an empty packet, and *TCPAckOptionsEncap*, *TCPAckEncap* generate the TCP header of a regular ACK packet. *DecTCPSeqNo* then decrements the TCP sequence number to trigger an ACK back from the other host. Finally, delayed ACKs are sent for every pair of received data packets unless a 500 ms timeout elapses. In this case, a regular ACK is sent using the modules described above.

3.5 Customization and Element Reuse

cTCP modularity enables code reuse and fine-grained customization of the network stack. For instance, TCP reliability can be disabled by simply removing *TCPEnqueue4RTX* from the configuration file. In this section, we present concrete examples to showcase the benefits of our modular TCP implementation.

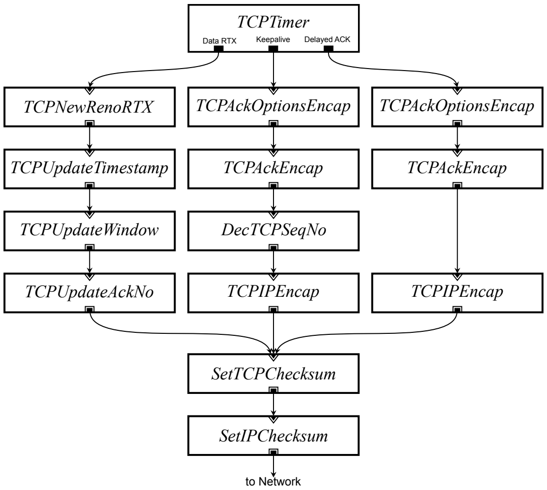


Figure 4: cTCP configuration for TCP timers.

Building a traffic generator that emulates several TCP flows concurrently sending at a constant rate is straightforward with ClickNF elements. The *TCPInfo* element is inserted in the configuration file and initialized with the corresponding TCBs. Data packets are generated and shaped at a constant rate by regular Click elements, *InfiniteSource* and *Shaper*, and then forwarded to a new element, *TCBSelector*, that randomly associates the packet to an existing TCB using ClickNF annotations. Afterwards, packets go through ClickNF elements such as *TCPAckEncap*, *TCPIPEncap* (plus optionally *SetTCPChecksum*, *SetIPChecksum*) to fill IP and TCP headers before being forwarded to an I/O element.

Moreover, per-flow congestion control can be used to ensure that specific traffic classes are processed using appropriate algorithms. Implementing such a feature in a monolithic OS network stack (e.g., Linux kernel one) is, however, quite complicated. Due to its modularity, ClickNF allows the definition of per-flow congestion control by simply inserting a *Classifier* element that modifies the behavior of cTCP for specific flows.

Finally, changing TCP New Reno to match data center TCP (DCTCP) [6] is as simple as adding a new *DCTCP-ProcessECN* element right after *TCPProcessAck* (Figure 2). This element modifies the TCP window behavior in presence of explicit congestion notification. Similarly, the introduction of a new congestion control algorithm, such as BBR [14], requires the development of few additional elements of low complexity.

4 Modular application

ClickNF enables the development of modular applications on top of cTCP. L7 network functions can be implemented using several flow-oriented elements, enabling the programmer to decouple packet processing from application state management logic. To do so, ClickNF separates network and application execution contexts in order to allow applications to block their execution when waiting for I/O operations. ClickNF also provides two fundamental building blocks, *i.e.,* socket I/O multiplexing and SSL/TLS termination elements, that enable rapid composition of complex and customized L7 functions.

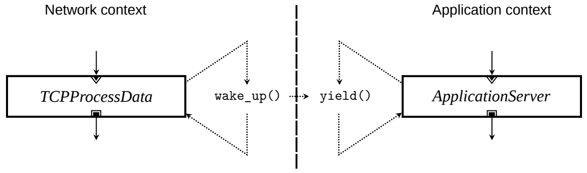


Figure 5: Context switch: *TCPProcessData* reschedules a blocked task waiting on I/O.

4.1 Blocking Tasks

ClickNF implements blocking I/O to provide developers with a broader range of I/O options. In Click, *tasks* are element-defined functions that require frequent CPU scheduling, and initiate a sequence of packet processing in the configuration graph. We introduce the concept of *blocking tasks*, which can yield the CPU if a given condition does not hold, e.g., an I/O request cannot be promptly completed. When rescheduled, the task resumes exactly where it left off, providing applications with an illusion of continuous execution. Blocking tasks are backward compatible with regular tasks, and require no modifications to the Click task scheduler.

Context switching between tasks is light-weight, saving and restoring registers required for task execution. ClickNF uses low-level functions to save and restore the tasks context, just as in POSIX threads. Differently than POSIX threads, however, ClickNF has access to the Click task scheduler and relies on cooperative, as opposed to preemptive, scheduling. ClickNF uses the Boost library to perform context switches in a handful of CPU cycles, *i.e.,* ≈20 cycles in x86_64 (few nanoseconds).

4.2 Network and Application Contexts

ClickNF uses blocking tasks to separate network and application execution contexts. The network context is active during packet reception (cf. Figure 3) and timeouts (cf. Figure 4), and runs through regular Click tasks. The application context, in contrast, is active during application processing and packet transmission (cf. Figure 2) and runs through blocking tasks.

A blocked application is scheduled when the event it is waiting on occurs, e.g., a task blocked on `accept` . Figure 5 presents an example of an application task being rescheduled by an event. In the example, *ApplicationServer* calls `epoll_wait` to monitor a group of active file descriptors. Since no one is ready to perform I/O, it calls `yield` to save the current context and unschedule the task. Later on, when a data packet is received, *TCPProcessData* checks if the application task is waiting for data packets and calls `wake_up` to reschedule it. The event is then inserted into a per-core event queue that stores the events that occurred for the sockets monitored by *ApplicationServer* (*i.e.,*different applications have separate event queues). When the application task is executed, to amortize the cost of the context switch, a batch of events is handled before the network context is re-scheduled.

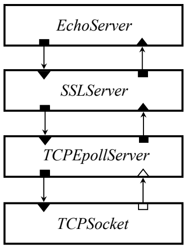


Figure 6: Configuration graph of a modular echo server in ClickNF that uses SSL/TLS encryption.

In ClickNF we specify a list of events needed to manage cTCP states and error conditions. For instance, events are generated when the accept queue becomes non-empty, or when the connection is established to wake up application tasks waiting on these conditions. Similarly, events are also generated when the TX queue becomes non-full, the RX queue becomes non-empty, and also when the RTX queue becomes empty. Other events signal that the remote peer wants to close the connection, the connection was closed, or an error occurred (e.g., a reset or timeout). Despite this fine grain event characterization, to remain compliant with the original epoll API, we map cTCP events to standard `EPOLLIN`, `EPOLLOUT`, and `EPOLLERR` events.

4.3 Application Building Blocks

To simplify application-level programming and promote code reuse, ClickNF provides four building blocks useful for practically relevant network functions. Such building blocks exchange control information with application layer elements using packet annotations. In this way, application elements are informed about the socket file descriptor to which the packets belongs and about new or closed connections. This allows them to efficiently multiplex data between different applications and multiple sockets in both directions.

The first application-layer building block, *TCPEpollServer*, implements an epoll server concealing the complexity of cTCP event handling and can be used to rapidly implement server-side network functions. Similarly, *TCPEpollClient* implements an epoll client to multiplex outgoing connections. Finally, *SSLServer* and *SSLClient* provide SSL/TLS encryption through the OpenSSL library, and may be used to implement network function that require end-to-end encryption. Application data enters in an input port of *SSLServer* as plaintext and leaves its output port as ciphertext; received packets take the reverse path to decrypt ciphertext into plaintext.

Figure 6 shows an echo server using SSL/TLS encryption as a straightforward example of a modular application assembled from ClickNF building blocks. Upon reception at *TCPEpollServer*, packets are decrypted by *SSLServer* and forwarded upstream. *EchoServer* is a stateless application that simply redirects the received data back to the client. On the way back, *SSLServer* encrypts the data and forwards it downstream to *TCPEpollServer*. This simple example shows how ClickNF enables a large number of possibilities for customizing the entire network stack of an application since any of its building blocks can be easily disabled or rewired. For instance, the echo server in Figure 6 can easily disable SSL/TLS for selected flows by introducing a classifier element into the configuration graph, without any change to the element that implements the application logic.

5 Implementation

ClickNF benefits from several improvements that the Click codebase received over time, such as fast packet I/O and multicore, besides introducing a brand new timer subsystem that copes with the scaling requirements of TCP support. This section highlights some notable technical details that characterize ClickNF implementation.

5.1 Packet I/O

Similarly to Fastclick [9], ClickNF provides fast packet I/O by using DPDK [33] to directly interface with network cards from user space. For packet reception, the DPDK element continuously poll the NIC to fetch received packet batches. In order to amortize the PCIe overhead, the DPDK element waits for a batch of 64 packets before transmitting them. To avoid head-of-line blocking and reduce latency, batches are transmitted after at most 100 μs. When needed, ClickNF performs batch processing (*i.e.,* Click elements process packet batches – implemented through packets' linked lists – instead of single packets) to optimize CPU cache performance. Also in this case, a batch is forwarded downstream after 100 μs even if it is not complete.

As in Fastclick, we modify the Click packet data structure to be a wrapper around a DPDK memory buffer (`mbuf`) to avoid additional memory allocation and copy operations. Each packet has a fixed size of 8 KB and consists of four sections, namely, the `mbuf` structure itself, packet annotations, headroom, and data. DPDK uses the `mbuf` for packet I/O whereas ClickNF uses annotations to store packet metadata (e.g., header pointers) and the headroom space to allow elements to prepend headers. Applications allocate a packet by filling only the data portion before pushing the packet down to lower layers.

Notice that, differently form Fastclick, we use a single element for both input and output operations in order to simplify the configuration. Moreover, ClickNF can also leverage common NIC features to perform flow control, TCP/IP checksum offloading, TCP segmentation (TSO), and large receive offloading (LRO) using hardware acceleration. Flow control prevents buffer overflows by slowing down transmitters when the RX buffer in the NIC becomes full. TCP/IP checksum offloading allows the NIC to compute header checksums in hardware. TSO segments a large packet into MTU-sized packets, whereas LRO aggregates multiple received packets into a large TCP segment before dispatching it to higher layers. All of these features can be toggled in ClickNF at run-time.

5.2 Multicore Scalability

Multithreading is implemented to exploit the processing power in multicore CPUs and improve scalability. We design per-core lock-free data structures aiming for high performance. Each core maintains dedicated packet pools, timers, transport, and application layer data structures. Receive Side Scaling (RSS) is used to distribute packets to different cores according to their flow identifier, *i.e.,* flow 5-tuple. Each DPDK thread is pinned to a CPU core and provided with a TX and a RX hardware queue at the NIC, preserving flow-level core affinity.

To avoid low-level CPU synchronization primitives each core maintains separate cache-aligned data structures. In case of multi-connection dependency, such as a proxy server establishing a connection on behalf of a client, the source port of the new outgoing connection is selected such that RSS maps it to the same core of the original connection, thus avoiding locks at the application level. Finally, each application-layer network function is spawned on multiple cores so that flows directed can be handled entirely on a specific core.

5.3 Timer Subsystem

Click implements its timer subsystem using a priority queue in which the root node stores the timer closer to expire. Given that TCP timers are often canceled before expiration, we implement a timing wheel scheduler for efficiency [47]. Its key advantage is that timing wheels schedule and cancel timers in $O(1)$, as opposed to $O(\log n)$ in priority queues currently used in Click.

A timing wheel is composed of n buckets, an index b, a timestamp t, and a tick granularity g (e.g., 1 ms). The timestamp t keeps the current time and the index b, points to its corresponding bucket. Each bucket contains timers expiring in the future, such that bucket b contains timers expiring within $[t, t+g)$, and so on. To schedule a timer, we must first find its corresponding bucket. For an expiration time e in the interval $t \leq e < t + ng$, its bucket index is computed as $\lfloor (e-t)/g \rfloor$. Each bucket contains a doubly linked list to store the timers expiring within the same interval. Therefore, once the index is computed, the timer is inserted at the end of the list of the bucket. To cancel a timer, the timer is just removed from the doubly linked list. Both operations are done in $O(1)$ with simple modulo operation to compute the bucket index b.

6 Evaluation

In this section, we evaluate ClickNF with three goals: (i) evaluate its performance through a series of microbenchmarks; (ii) compare it against Linux and state-of-the-art user space stacks; and (iii) showcase the usage of ClickNF and its performance when building network functions. Our evaluation setup consists of 3 machines with Intel Xeon® 40-core E5-2660 v3 2.60GHz processors, 64 GB RAM, each equipped with an Intel® 82599ES network card containing two 10 GbE interfaces. The machines run Ubuntu 16.10 (GNU/Linux 4.4.0-51-generic x86_64), Click 2.1, and DPDK 17.02.

6.1 Microbenchmarks

We start by analyzing individual aspects of our system using microbenchmarks, including packet I/O throughput, the cost of modularity, and the impact of hardware offloading. We then evaluate two applications, namely bulk transfer and echo server, to understand the system performance in common scenarios. Unless otherwise specified, the experiments presented in this section are performed on single-core ClickNF instances.

Packet I/O and the cost of modularity: To evaluate the throughput of our DPDK element, we run a set of tests with the DPDK traffic generator (DPDK-TG) [25] on one side and ClickNF on the other one. For ClickNF, we use four configurations that respectively generate and immediately discard (no I/O), receive (RX), forward (FW), and transmit (TX) 64-byte packets. Finally, we evaluate the cost of modularity by adding *PushNull* elements that receive packets on the input port and send them on the output port without doing anything else.

Figure 7 presents the average throughput for the different scenarios with a series of *PushNull* elements. Without I/O, ClickNF throughput is limited by the CPU at 43 Mpps. Increasing the number of elements increases the time spent by a packet inside the Click graph, considerably reducing the system throughput. For the RX, FW, and TX scenarios, the throughput is limited by the NIC line rate at 14.88 Mpps. ClickNF is still able to sustain the line rate with up to 15–20 *PushNull* elements. At this point, the throughput is limited by CPU and decreases further as more elements are placed in the configuration.

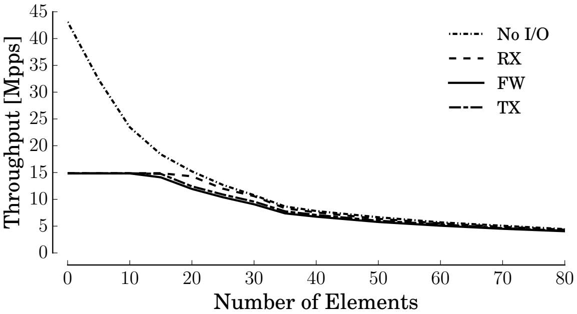


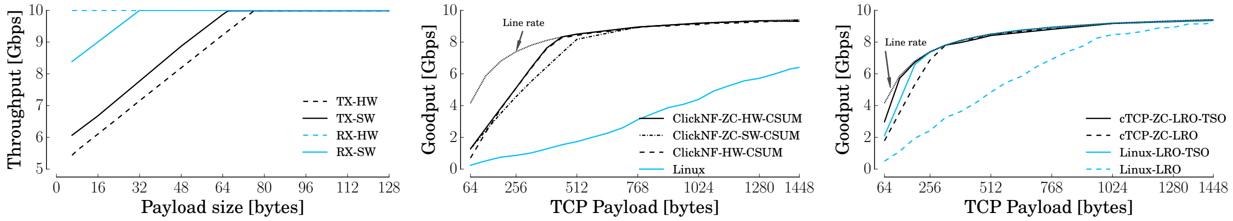
Figure 7: Throughput with 64-byte packets and increasing number of Click elements in different scenarios.

However, using packet batches between Click elements (Section 5.1) reduces the cost of modularity by improving instruction and data cache utilization. Indeed, when processing batches instead of single packets we experimentally evaluate that ClickNF to sustain line rate with up to 150 *PushNull* elements in RX, TX, and FW configurations. In ClickNF we adopt batch processing with batches of 32 packets to improve performance.

Checksum offloading: To evaluate hardware checksum offloading in our DPDK module, we measure the throughput using software or hardware checksum computation-verification. As in the previous tests, we run the DPDK-TG on one side and ClickNF on the other. We then use two different configurations for transmitting (TX) and receiving (RX) packets belonging to a single TCP flow. For transmissions, ClickNF computes header checksums before transmitting the packets to the traffic generator. For receptions, ClickNF verifies whether the checksums are correct before discarding the packet. Both operations are performed in hardware or in software.

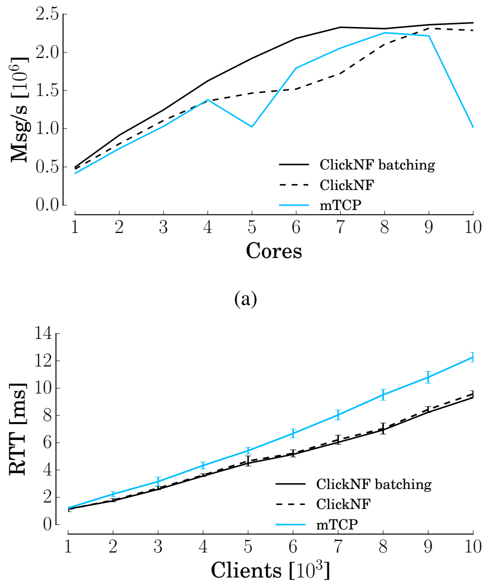
Figure 8a shows the results for TX and RX with increasing payload size (6–128 bytes). As expected, offloading checksum verification provides significant performance benefits and allows ClickNF to sustain line rate even when receiving small packets. Surprisingly, TX checksum computation in software is significantly better than in hardware. This is because modern CPUs are very efficient when performing sequential operations on cached memory, and sometimes even faster than dedicated hardware. However, as we see in the next experiments, this only holds because the CPU is underloaded.

Bulk transfer: To validate ClickNF in a more realistic scenario we execute a bulk transfer of a 20 GB file from a client to a server. Moreover, we evaluate the performance of TCP segment offloading (TSO) and large received offloading (LRO), as well as equivalent implementations in software Click elements. For comparison, we use iperf [4] running on top of Linux network.



(a) (b) (c)

Figure 8: (a) TCP throughput with and without TCP/IP checksum offloading. (b) TCP goodput in a bulk transfer with increasing TCP payload size. (c) TCP goodput in a bulk transfer with TSO and LRO enabled.



(a)

(b)

Figure 9: Comparison between ClickNF and mTCP. (a) Echo server message rate with increasing number of cores. (b) RTT with increasing number of TCP clients.

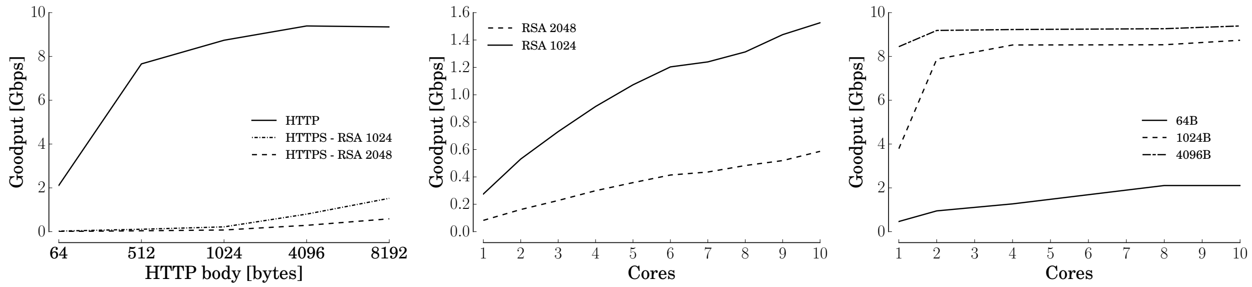
Figure 8b shows the TCP goodput for increasing payload sizes for ClickNF and Linux. For ClickNF, we use the socket API with HW checksum computation-verification (ClickNF-HW-CSUM), the zero-copy API with either software checksum computation-verification (ClickNF-ZC-SW-CSUM) or hardware checksum offloading (ClickNF-ZC-HW-CSUM). For Linux, we use iperf for tests with hardware checksum enabled. ClickNF significantly outperforms Linux and achieves line rate for TCP payloads larger than 448 bytes when the packet header overhead is smaller. For 64-byte payload, the zero-copy API provides roughly 50% throughput improvement over the socket API. This occurs because, with larger packets, the number of calls to `memcopy` and `recv` decreases, limiting the advantage of zero copy. In the following tests, we use the zero-copy API, as it delivers better performance with respect to the socket API.

Unlike what is observed in reception, TX checksum offloading (ClickNF-ZC-HW-CSUM) provides significant benefits with respect to software (ClickNF-ZC-SW-CSUM). For computationally intensive workloads, TX checksum offloading prevents the CPU from spending precious cycles in checksum computation-verification.

Figure 8c presents the results with TSO and LRO enabled. ClickNF outperforms Linux, specially for small payloads suggesting that the Linux stack is particularly inefficient for small packets, as reported in [20, 35, 10]. ClickNF achieves line rate for TCP payloads larger than 128 bytes while Linux have similar performance with TSO and LRO for TCP payloads larger than 192 bytes. In the following, we always enable LRO and TSO to amortize segmentation and reassembly cost.

Echo server: We also evaluate ClickNF in the presence of short TCP connections and compare its performance against mTCP [20], a user-space network stack with particular focus on performance and with similar goals to ClickNF. To evaluate multicore scalability, we run an echo server on top of both stacks. Clients running in two separate 8-core ClickNF instances connect to the server, send a 64-byte message, and wait for the echo reply. When the client receives the message back, it resets the connection to avoid port exhausting. The client then repeats the operation and measures the message rate. To provide a fair comparison against mTCP, we disable delayed ACKs that, when enabled, decrease the overhead and increase the overall throughput. Figure 6 depicts the configuration graph of the echo server used in this test, except for the *SSLServer* element that is not included.

First we measure the rate obtained by ClickNF and mTCP with a single core. ClickNF provides high throughput, 0.5×10^6 Msg/s, when using DPDK packet I/O. Compared to legacy Click elements for packet I/O, 0.169×10^6 Msg/s, (*i.e.,* using PCAP library linked to *FromDevice*), ClickNF provides 4x higher throughput. This shows how important kernel bypass is when enabling zero-copy packet processing at user space. Additionally, ClickNF outperforms mTCP, 0.415×10^6 Msg/s, by approximately 20%.



(a) HTTP cache - 10 cores with/without SSL/TLS

(b) HTTP cache - with SSL/TLS

(c) Proxy

Figure 10: Average goodput of the ClickNF modular HTTP(S) cache server.

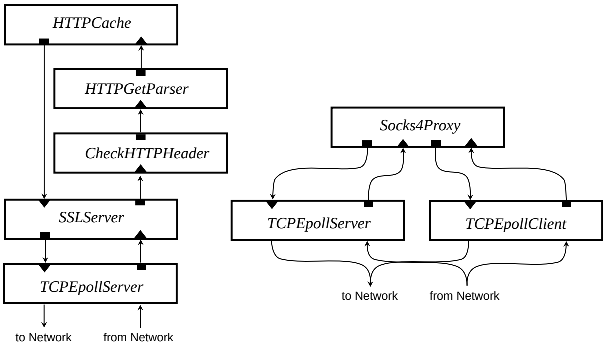


Figure 11: Configuration graphs for an HTTP cache server with SSL/TLS and for a SOCKS4 Proxy.

Figure 9a presents the message rate obtained with ClickNF and with mTCP for increasing number of cores. Despite its modularity, ClickNF provides equivalent performance to mTCP up to 4 cores, and scales slower for more cores. As observed earlier, modularity has a cost, but can be amortized with packet batches. In this case, Click elements receives packet batches instead of single packets hence optimizing CPU instruction and data cache usage. We enable batching in ClickNF for L2–L3 operations to avoid packet reordering issues at L4 and repeated the echo server experiment. Figure 9a shows that, ClickNF outperforms mTCP and achieves line rate with 7 cores. Results with hyperthreading (not reported here) show higher throughput, reducing the number of cores required to saturate the link to 4.

Since ClickNF employs batching at all levels, the risk is that RTT might be undesirably long. Figure 9b shows the RTT experienced by ClickNF and mTCP in the single core echo server test with increasing number of concurring clients. Due to the usage of batch timeouts, the latency introduced with increasing number of clients is limited and lower when compared to mTCP.

6.2 Modular Network Functions

To evaluate ClickNF performance and show the benefits of its modularity, we build two sample applications.

HTTP(S) cache: Figure 11 depicts the configuration graph used to deploy an HTTP cache server. Using the basic building blocks provided by ClickNF, application logic is implemented with three simple elements.

To evaluate the performance of our modular HTTP cache server, we run tests with SSL/TLS termination using self-signed certificates for 1024- or 2048-byte RSA keys. Clients running in two 8-core ClickNF instances first connect to the server and then issue HTTP GET requests for web pages of size 64–8192 bytes, stored in the HTTP cache server's main memory. The server responds to the requests and then closes the TCP connection.

Figure 10a,10b presents the goodput of the ClickNF HTTP cache server with and without SSL/TLS termination. With unencrypted HTTP traffic running on 10 cores (cf. Figure 10a), ClickNF achieves high goodput for small HTTP pages, and scales linearly with bigger page size. With SSL/TLS termination, the goodput drops to a maximum of $\approx$ 1.6 Gbps for 1024-byte keys and 10 cores. This is due to the complexity of public-key RSA cryptographic operations during SSL/TLS handshake [15]. This overhead, however, can be alleviated by delegating such operations to GPUs [27, 48].

SOCKS4 proxy: Socket Secure (SOCKS) is a protocol for enabling client-server communication through a proxy. Starting from a basic SOCKS4 proxy implementation written in C, we built a modular SOCKS4 proxy composed by three elements namely *Socks4Proxy*, and ClickNF building blocks *TCPEpollServer* and *TCPEpollClient*. Figure 11 depicts the high-level configuration graph for the proxy. Notice that, due to ClickNF L7 modularity, the SOCKS4 proxy graph can be easily modified to introduce additional functions (e.g., firewall, SSL encryption) right before the paths connecting *TCPEpollServer* and *Socks4Proxy*.

To evaluate the performance of our modular SOCKS4 proxy, we run a simple test similar to the one presented for the HTTP cache. In this case, clients connect to the SOCKS4 proxy which opens a new connection toward the HTTP cache. Once the connection is established, the client requests an HTTP page of fixed size and then resets the connection. Figure 10c presents the goodput of the ClickNF SOCKS4 proxy with increasing number of cores and variable HTTP body sizes. Similarly to the HTTP cache experiment, when the HTTP message is small the overhead (connection establishment and SOCKS protocol) is significant and prevents the system from achieving a higher goodput. For larger page sizes, the overhead decreases and the proxy is able to achieve close to line rate using just two CPU cores.

7 Related Work

Click [29] and its modular data plane have been improved and extended in multiple directions over almost two decades. For instance, Routebricks [18] parallelizes routing functionality across and within software routers building on top of Click to scale its performance. Recently, Fastclick [9] introduced high-speed packet I/O such as DPDK and netmap [33, 44] in Click.Moreover, GPU offloading is also proposed in [28, 46] to increase throughput beyond CPU capabilities. Similarly, ClickNP [32] provides Click-like programming abstractions to enable the construction of FPGA-accelerated network functions. ClickNF is orthogonal to such extensions and enables the modular composition of L2–L7 stacks, bridging Click's L2-L3 packet processing with L4 flow processing and L7 modular applications.

Click inspired other modular network function frameworks [8, 12, 40]. These systems mainly focus on control plane operations, such as data plane element placement, network function scaling, and traffic steering. For the data plane, FlowOS [11] is proposed as a middlebox platform that enables flow processing, but without TCP support. CoMb [45] and xOMB [7] use Click to consolidate middleboxes through the composition of different L7 elements. Both rely on the OS for packet I/O and transport layer, reducing customization and performance. Frameworks to enable stack customization of L2–L7 are proposed in [42, 26]. In [42], authors introduce an overview of the control and data planes of a modular architecture, with focus on hardware acceleration. In [26], the design of a modular middlebox platform based on mTCP [20] is presented. Instead of redesigning a framework with Click-like abstractions and/or providing new event-driven domain-specific APIs, ClickNF introduces L2–L7 modularity in Click to expose and exploit its modularity, as well as benefiting from existing Click extensions and contributions by the community.

Network stacks were proposed to overcome the I/O inefficiencies of OS [10, 43, 20, 35, 23, 49, 16]. IX [10] separates the control plane and data plane processing. Arrakis [43] is a customized OS that provide applications with direct access to I/O devices, allowing kernel bypass for I/O operations. mTCP [20] is a user-level TCP implementation proposed for multicore systems. It provides a socket API for application development supporting L2–L4 zero copy. In a different spirit, Stackmap [49] provides packet I/O acceleration to TCP kernel implementation obtaining better performance compared to Linux TCP. Sandstorm [35] proposes a specialized network stack with zero-copy APIs merging application and network logics. Similarly to ClickNF, Modnet [41] has a modular approach for providing customizable networking stack, but modularity is limited to L2–L4. Few other efforts [3, 5, 2, 16, 13] also provide efficient, sometimes modular, networking stacks but cannot completely benefit of L2–L7 modularity provided by ClickNF.

Our previous workshop paper [31] focused on providing an initial architecture for a modular TCP implementation in Click. ClickNF extends it in several directions. For instance, ClickNF takes advantage of hardware offloading, multicore scalability, timing wheels, and an epoll-based API to improve performance. Application level modularity and SSL/TLS termination provide building blocks for novel network functions to be deployed with little effort. We hereby propose a more comprehensive picture of ClickNF's performance, flexibility, and ease of use.

8 Conclusions

The advent of NFV gives us a different perspective on the way application servers and middleboxes can be implemented. ClickNF enables the composition of high-performance network functions at all layers of the network stack and opens up its inner workings for the benefit of developers. In this paper, we illustrated and benchmarked several concrete examples where ClickNF can be used to accelerate network function development by enabling fine-grained code reuse, and highlighted ClickNF's good scaling properties and a reasonable price of modularity, which arguably outweigh many of the hurdles in network function development. The ClickNF source code is available for download at [21].

Acknowledgments

We would like to thank our anonymous reviewers that provided several advices to improve and further develop our work as well as Fabio Pianese for the fruitful discussions and for his countless paper revisions.

References

[1] The Linux Kernel. /urlhttps://www.kernel.org/.

[2] Open Fast path, 2015. http://www.openfastpath.org/.

[3] 6windgate, 2017. http://www.6wind.com/products/6windgate/.

[4] Iperf2, 2017. https://sourceforge.net/projects/iperf2/.

[5] The Fast Data Project FD.io, 2017. https://fd.io/.

[6] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference* (2010), SIGCOMM '10.

[7] ANDERSON, J. W., BRAUD, R., KAPOOR, R., PORTER, G., AND VAHDAT, A. xOMB: Extensible Open Middleboxes with Commodity Servers. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (2012), ANCS '12, ACM.

[8] ANWER, B., BENSON, T., FEAMSTER, N., AND LEVIN, D. Programming Slick Network Functions. In *Proc. 1st ACM SIGCOMM Symposium on Software Defined Networking Research* (2015), SOSR '15.

[9] BARBETTE, T., SOLDANI, C., AND MATHY, L. Fast Userspace Packet Processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (2015), ANCS '15, IEEE Computer Society.

[10] BELAY, A., PREKAS, G., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proc. 11th USENIX OSDI* (2014).

[11] BEZAHAF, M., ALIM, A., AND MATHY, L. FlowOS: A Flow-based Platform for Middleboxes. In *Proceedings of the 2013 Workshop on Hot Topics in Middleboxes and Network Function Virtualization* (2013), HotMiddlebox '13, ACM.

[12] BREMLER-BARR, A., HARCHOL, Y., AND HAY, D. Openbox: A software-defined framework for developing, deploying, and managing network functions. In *Proc. of the 2016 ACM SIGCOMM Conference* (2016).

[13] BRIDGES, P. G., WONG, G. T., HILTUNEN, M., SCHLICHTING, R. D., AND BARRICK, M. J. A configurable and extensible transport protocol. *IEEE/ACM Transactions on Networking 15*, 6 (2007), 1254–1265.

[14] CARDWELL, N., CHENG, Y., GUNN, C. S., YEGANEH, S. H., AND JACOBSON, V. Bbr: Congestion-based congestion control. *Queue 14*, 5 (2016), 50:20–50:53.

[15] COARFA, C., DRUSCHEL, P., AND WALLACH, D. S. Performance analysis of TLS web servers. *ACM Transaction of Compututer System* (2006).

[16] CONDIE, T., HELLERSTEIN, J. M., MANIATIS, P., RHEA, S., AND ROSCOE, T. Finally, a use for componentized transport protocols. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (2005), HotNets-IV.

[17] CRONKITE-RATCLIFF, B., BERGMAN, A., VARGAFTIK, S., RAVI, M., MCKEOWN, N., ABRAHAM, I., AND KESLASSY, I. Virtualized congestion control. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), SIGCOMM '16.

[18] DOBRESCU, M., EGI, N., ARGYRAKI, K., CHUN, B.-G., FALL, K., IANNACCONE, G., KNIES, A., MANESH, M., AND RATNASAMY, S. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (2009), SOSP '09, ACM.

[19] ETSI. Network Function Virtualization. *SDN & OpenFlow World Congress* (2014).

[20] EUNYOUNG, J., SHINAE, W., MUHAMMAD, J., HAEWON, J., SUNGHWAN, I., DONGSU, H., AND KYOUNGSOO, P. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proc. 11th USENIX NSDI* (2014).

[21] GALLO, M., AND LAUFER, R. The ClickNF framework. Available on https://github.com/nokia/ClickNF, 2017.

[22] HE, K., ROZNER, E., AGARWAL, K., GU, Y. J., FELTER, W., CARTER, J., AND AKELLA, A. Ac/dc tcp: Virtual congestion control enforcement for datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), SIGCOMM '16.

[23] HRUBY, T., GIUFFRIDA, C., SAMBUC, L., BOS, H., AND TANENBAUM, A. S. A NEaT Design for Reliable and Scalable Network Stacks. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies* (2016), CoNEXT '16, ACM.

[24] HWANG, J., RAMAKRISHNAN, K. K., AND WOOD, T. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. *IEEE Transactions on Network and Service Management* (2015).

[25] INTEL. DPDK Traffic Generator, 2017. http://dpdk.org/browse/apps/pktgen-dpdk/.

[26] JAMSHED, M. A., MOON, Y., KIM, D., HAN, D., AND PARK, K. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *Proc. of 14th USENIX NSDI* (2017).

[27] JANG, K., HAN, S., HAN, S., MOON, S., AND PARK, K. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proc. 8th USENIX NSDI* (2011).

[28] KIM, J., JANG, K., LEE, K., MA, S., SHIM, J., AND MOON, S. NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), EuroSys '15, ACM.

[29] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click Modular Router. *ACM Transaction of Computer System* (2000).

[30] LANGLEY, A. E. A. The quic transport protocol: Design and internet-scale deployment. In *Proc. of the 2017 ACM SIGCOMM Conference* (2017).

[31] LAUFER, R., GALLO, M., PERINO, D., AND NANDUGUDI, A. CliMB: Enabling network function composition with Click middleboxes. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization* (2016), HotMIddlebox '16, ACM.

[32] LI, B., TAN, K., LUO, L. L., PENG, Y., LUO, R., XU, N., XIONG, Y., CHENG, P., AND CHEN, E. ClickNP: Highly flexible and high performance network processing with reconfigurable hardware. In *Proc. of the 2016 ACM SIGCOMM Conference* (2016).

[33] LINUX FOUNDATION. DPDK, 2017. http://dpdk.org.

[34] MANCO, F., LUPU, C., SCHMIDT, F., MENDES, J., KUENZER, S., SATI, S., YASUKATA, K., RAICIU, C., AND HUICI, F. My vm is lighter (and safer) than your container. In *Proceedings of the Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 218–233.

[35] MARINOS, I., WATSON, R. N., AND HANDLEY, M. Network Stack Specialization for Performance. In *Proc. of the 2014 ACM SIGCOMM Conference* (2014), SIGCOMM '14, ACM.

[36] MARTINS, J., AHMED, M., RAICIU, C., OLTEANU, V., HONDA, M., BIFULCO, R., AND HUICI, F. ClickOS and the Art of Network Function Virtualization. In *Proc. 11th USENIX NSDI* (2014).

[37] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.* (2008).

[38] Narayan, A., Cangialosi, F., Goyal, P., Narayana, S., Alizadeh, M., and Balakrishnan, H. The case for moving congestion control out of the datapath. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (2017), HotNets-XVI.

[39] Niu, Z., Xu, H., Han, D., Cheng, P., Xiong, Y., Chen, G., and Winstein, K. Network stack as a service in the cloud. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (2017), HotNets-XVI.

[40] Palkar, S., Lan, C., Han, S., Jang, K., Panda, A., Ratnasamy, S., Rizzo, L., and Shenker, S. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP '15, ACM.

[41] Pathak, S., and Pai, V. S. Modnet: A modular approach to network stack extension. In *Proc. 12th USENIX NSDI* (2015).

[42] Perino, D., Gallo, M., Laufer, R., Houidi, Z. B., and Pianese, F. A Programmable Data Plane for Heterogeneous NFV Platforms. In *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)* (2016).

[43] Peter, S., Li, J., Zhang, I., Ports, D. R. K., Woos, D., Krishnamurthy, A., Anderson, T., and Roscoe, T. Arrakis: The Operating System is the Control Plane. In *Proc. 11th USENIX OSDI* (2014).

[44] Rizzo, L. netmap: A Novel Framework for Fast Packet I/O. In *Proc. of the 2012 USENIX Annual Technical Conference* (2012).

[45] Sekar, V., Egi, N., Ratnasamy, S., Reiter, M. K., and Shi, G. Design and Implementation of a Consolidated Middlebox Architecture. In *Proc. of 9th USENIX NSDI* (2012).

[46] Sun, W., and Ricci, R. Fast and Flexible: Parallel Packet Processing with GPUs and Click. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (2013), ANCS '13, IEEE Press.

[47] Varghese, G., and Lauck, A. Hashed and hierarchical timing wheels: Efficient data structures for implementing a timer facility. *IEEE/ACM Transactions on Networking* (1997).

[48] Varvello, M., Laufer, R., Zhang, F., and Lakshman, T. Multilayer packet classification with graphics processing units. *IEEE/ACM Transactions on Networking* (2016).

[49] Yasukata, K., Honda, M., Santry, D., and Eggert, L. Stackmap: Low-latency networking with the OS stack and dedicated nics. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (2016).