Microboxes: High Performance NFV with Customizable, Asynchronous TCP Stacks and Dynamic Subscriptions

Guyue Liu*, Yuxin Ren*, Mykola Yurchenko*,
K.K. Ramakrishnan†, Timothy Wood*
*George Washington University, †University of California, Riverside

ABSTRACT

Existing network service chaining frameworks are based on a "packet-centric" model where each NF in a chain is given every packet for processing. This approach becomes both inefficient and inconvenient for more complex network functions that operate at higher levels of the protocol stack. We propose Microboxes, a novel service chaining abstraction designed to support transport- and application-layer middleboxes, or even end-system like services. Simply including a TCP stack in an NFV platform is insufficient because there is a wide spectrum of middlebox types–from NFs requiring only simple TCP bytestream reconstruction to full endpoint termination. By exposing a publish/subscribe-based API for NFs to access packets or protocol events as needed, Microboxes eliminates redundant processing across a chain and enables a modular design. Our implementation on a DPDK-based NFV framework can double throughput by consolidating stack operations and provide a 51% throughput gain by customizing TCP processing to the appropriate level.

CCS CONCEPTS

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

KEYWORDS

Middleboxes, NFV, Networking Stack, Service Chain

ACM Reference Format:
G. Liu, Y. Ren, M. Yurchenko, K.K. Ramakrishnan, T. Wood. 2018. Microboxes: High Performance NFV with Customizable, Asynchronous TCP Stacks and Dynamic Subscriptions. In *Proceedings of ACM SIGCOMM 2018 Conference (SIGCOMM '18)*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3230543.3230563

SIGCOMM '18, August 20–25, 2018, Budapest, Hungary
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5567-4/18/08…$15.00
https://doi.org/10.1145/3230543.3230563

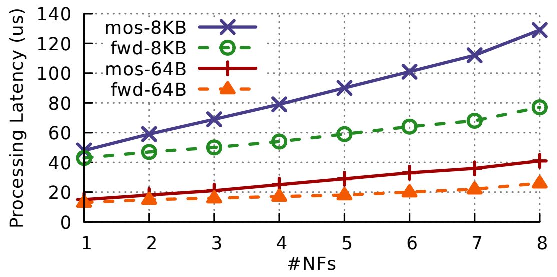


Figure 1: Repeated TCP stack processing in a chain of mOS NFs can cause unnecessarily high delay.

1 INTRODUCTION

Today's enterprise and wide-area networks are filled with middleboxes [27] providing a wide range of functionality from simple firewalls to complex Evolved Packet Core (EPC) functions in cellular networks. Network Function Virtualization (NFV) platforms provide high performance packet processing by leveraging kernel bypass I/O libraries such as DPDK [1] and netmap [25]. However, these systems are packet-centric: they focus on providing efficient movement of packets through a chain of network functions that operate on each packet as it arrives. While this model can make sense for simple layer-2/3 processing, it becomes inefficient and inconvenient when building more complex functions operating at higher levels of the protocol stack.

Network functions that operate at the transport layer need to perform additional processing such as TCP bytestream reconstruction. This is a relatively heavyweight function since it involves copying packet data into a buffer, an action that is avoided in many layer-2/3 middleboxes that rely on "zero-copy" to achieve high throughput. High performance, user-space TCP stacks [12, 13] can be used by NFs to simplify this operation, but these libraries must be used individually by each NF, resulting in redundant computation if a chain of functions each perform TCP processing.

To illustrate the high cost of redundant TCP processing, Figure 1 shows the processing latency for a chain of NFs that perform TCP bytestream reconstruction using mOS [12] or simply forward individual packets at layer 2 (fwd in figure); to maximize performance, each NF runs on its own core. As the chain length increases, the latency for the NFs performing TCP processing increases substantially compared to that for layer-2 NFs. Ideally, this added latency could be avoided by performing TCP processing only once, and then exposing the resulting stream to the sequence of NFs.

While consolidating the stack processing within a chain eliminates redundant work, supporting such an architecture in an NFV platform can be difficult since each chain may require different levels of TCP processing. Some flows may require full bytestream reconstruction, while others may only need simpler detection of TCP state transitions. Still other NFs may require complete TCP endpoint termination, for example to host a caching proxy. Supporting this spectrum requires a customizable TCP processing engine that only performs the necessary work for each flow of packets.

We have designed the Microboxes framework in order to provide the appropriate level of protocol processing without incurring redundant work for chains. Microboxes allow complex NFs to be broken into their composite parts, and for chains of functions to be flexibly and efficiently connected together. The Microboxes framework provides modular protocol processing engines that can be tuned to support only the features needed for a given flow. Providing this appropriate protocol functionality for individual flows requires us to go beyond just a packet-centric view of middlebox service chaining. Microboxes produce and consume events that may correspond to individual packet arrivals, protocol events, or application-level actions. Our platform makes the following contributions:

- A publish/subscribe architecture that provides convenient, higher level interfaces based on the flow of events rather than the flow of packets.
- A consolidated protocol stack that eliminates redundant computation in chains of NFs.
- A customizable stack to support a spectrum of middlebox types ranging from lightweight TCP state monitoring to proxies with full TCP termination.
- Techniques for asynchronous parallel processing across NFs and the stack to improve scalability while meeting consistency requirements.

We have implemented Microboxes using a high performance, DPDK-based NFV framework. Our evaluation shows that the Microboxes consolidated TCP stack doubles throughput and lowers latency by 20% compared to mOS when running a chain of 6 TCP-based NFs. Compared to the industry-standard HAProxy, a web proxy built on our platform provides between 30-51% throughput improvement by customizing stack processing e.g., using our TCP Splicer Stack that allows zero-copy, lightweight URL-based redirection, or using a full TCP Proxy Stack that supports arbitrary data transformations at higher cost. This allows NFs to choose the appropriate level of protocol processing for each flow's performance and functional requirements.

	Header	Payload	Example NF
Layer-2/3	RW	–	Firewall
TCP Monitor	R	R	IDS
TCP Splicer	RW	R	L7 Load Balancer
TCP Split Proxy	RW	RW	Proxy

Table 1: Different types of NFs require different levels of TCP processing with varying costs

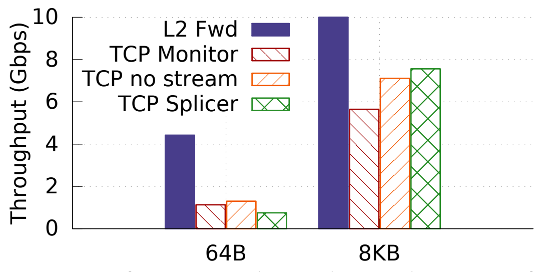


Figure 2: Performance depends on the type of stack processing needed and the incoming workload.

2 MIDDLEBOX STACK DIVERSITY

It is important to go beyond simply creating a "one size fits all" TCP stack to be used for all NFs—depending on the nature of the NF it may only require certain protocol events or may need unique types of processing. Flows with different requirements are likely to be consolidated on to a single NFV host. Table 1 enumerates a spectrum of common middlebox types with distinct protocol processing requirements. **Layer-2/3** NFs require no TCP state, but may require the ability to read and write packet headers, with no inspection of payload data. **TCP Monitor** middleboxes, such as an Intrusion Detection System (IDS), require TCP reconstruction and primarily monitor flow state; they may drop connections but do not need to modify individual packets. Next, **TCP Splicer** middleboxes must both read and write TCP header data in order to redirect traffic, for example a Layer-7 Load Balancer might perform a TCP handshake with a client and observe the content in an HTTP GET request before selecting a backend and handing off the connection to that server, without any modifications to the bytestream. Finally, middleboxes that perform **TCP Split Proxy** need complete control over the bytestream. For example, a split TCP-proxy might compress a server's replies before sending them over a second connection to the client.

The amount of work that must be performed for protocol processing in each of these cases can vary significantly. To demonstrate this, we evaluate the throughput of several middleboxes processing traffic from a web client requesting 64B or 8KB files from a server (Figure 2). The NFs are designed to perform minimal extra work beyond the requisite protocol processing. The Layer 2 Fwd NF simply forwards packets out the NIC, so it gets the highest throughput, note that the web server and client become the bottleneck for

64B requests. The TCP Monitor NF uses the mOS TCP middlebox library [12] to track connection state and perform flow reconstruction, while the "TCP no stream" NF is the same, but does not reconstruct the bytestream. Performing TCP processing substantially reduces throughput compared to simple forwarding, although the overhead is not as high if the NF does not need to track the bytestream. The TCP Splicer NF uses DPDK library and acts as a Splicer [9, 18], initially responding to the client, and then replaying the TCP handshake with a server. This has high overhead when requests are small (since extra work needs to be done to setup each new connection), but performs fairly well when the request size is large since it simply relays data packets after the connection is set up. These results show that an NFV platform running a variety of NF services is likely to require different types of stack processing for different flows.

This motivates our design of Microboxes to support flexible protocol stacks that can be customized on a per-flow basis to perform only the type of processing required by an NF.

3 MICROBOXES DESIGN

Monolithic NF architectures, illustrated in Figure 3(a), group functions together into a single address space, and are used in platforms such as BESS [11] and mOS [12]. This provides low overhead chaining since each NF is just a function call, but it requires NF implementations to be tightly coupled. Instead, Microboxes focuses on NFs built using a pipeline model, i.e., functions run as isolated processes or containers and packet data flows through a chain of service functions to achieve an overall application goal. The pipelined approach provides deployment benefits since NFs from different vendors can be easily grouped together to build complex behavior and can be elastically scaled out by adding more cores. However, existing pipeline-based NFV platforms such as OpenNetVM [29] or ClickOS [19] focus on moving packets, not higher level protocol stack events, as shown in Figure 3(b). Microboxes eschews the packet-centric design of prior platforms in order to extract common protocol processing functionality into a μStack, and expose a flexible μEvent interface between NFs to eliminate redundant work, while maintaining the deployment flexibility of the pipeline model.

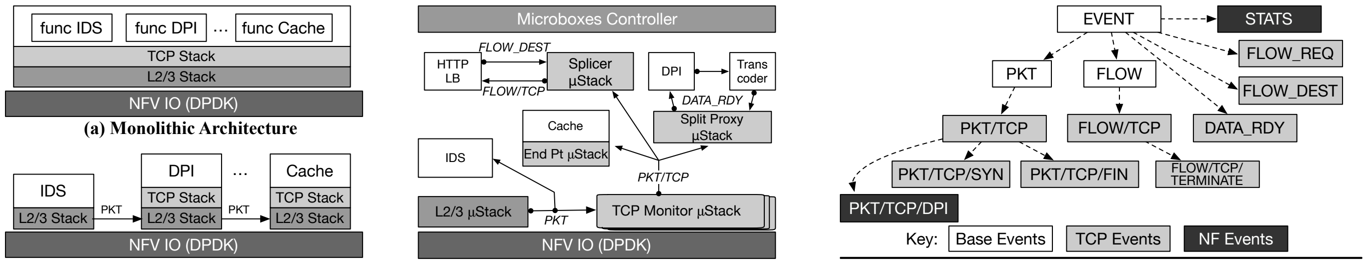
3.1 μEvents

The Microboxes event management system propagates events (rather than packets) through NFs to achieve the desired protocol- and application-layer processing. The μEvent messaging system is based around subscriber lists, which indicate the other NFs interested in events produced by a protocol stack or NF. Subscriptions can be filtered on a per-flow basis, and are managed by the Microboxes Controller.

Base Event Types: An event type defines a message format used to facilitate communication and determine how NFs and stacks can be interconnected. Events must follow well-defined data types to ensure that a subscriber will be able to interpret the message; we assume Microboxes developers share event IDs and data structure definitions for events they wish to propagate to other NFs. Figure 3(d) shows the hierarchy of event types defined by the base layer, TCP stack, and our sample NFs. Other NFs can extend these to create more complex messages. An `EVENT` is the simplest type on which others are based and is defined by fields indicating the event type and the publishing NF's ID. The `EVENT/PKT` type extends this data structure to include information about a specific packet and the flow state related to it. NFs that do not require every packet, but only need to be notified of state changes such as the start or end of a flow can subscribe to `EVENT/FLOW`. For brevity, we generally omit the `EVENT/` prefix. The `PKT` and `FLOW` based events provide flexibility, e.g., an NF might require knowledge of every packet related to a connection closing (`PKT/TCP/FIN`), or it might only care about the overall state change (`FLOW/TCP/TERMINATE`).

Event Hierarchy: An event type defines a messaging format similar to a communication protocol. A type can then be "extended" to create a new type that begins with all of the parent fields, and possibly adds extra ones, similar to layering an additional protocol header. For example, the `PKT/TCP` event indicates that a packet has been processed by a TCP stack, and adds fields for the TCP stack state to its parent event's data format. However, unlike layering protocol headers, child events are not required to add additional data fields to the parent type. For example, `PKT/TCP` is further extended to represent important TCP events, such as `PKT/TCP/SYN`, `PKT/TCP/FIN`, and `PKT/TCP/RETRANSMISSION`. In these cases the child events are used to provide a way for NFs to subscribe to the subset of the parent events that they care about. Thus the event hierarchy serves two purposes: it provides abstraction over data structures to allow more flexible type checking and it offers more flexible ways for NFs to filter the types of events they want to receive.

Event Subscription: Network functions and protocol stacks publish and subscribe to events to exchange information and trigger processing. When an NF starts, it can create a subscription by specifying an event type and a callback function handler. The hierarchy of types standardizes subscriptions so the Controller, described below, can perform type checking and determine how the new subscription can be connected to a publishing NF or stack component. The type relationship X/Y defines a strict hierarchy where X/Y is a child of X and includes all of its data fields and possibly more. Thus an NF that subscribes to messages of type X can also handle



(a) Monolithic Architecture

(b) Pipelined Architecture **(c) Microboxes Architecture** **(d) Microboxes Event Types**

Figure 3: (a) Monolithic architectures combine NFs as function calls in a single process, but this hurts deployment. (b) Pipelined NFs deployed as containers are easier to manage, but prone to redundant computation. (c-d) Microboxes customizes protocol processing in μStack modules and facilitates communication with a hierarchy of μEvent types.

messages of type X/Y, but not necessarily the opposite. Using this hierarchy provides a well-defined interface for type checking and filtering, compared to previous NFV packet pipeline frameworks where data moving between NFs has no defined format. The typing of ports are similar to Click platform [14], instead of having only pull and push ports, we extend it into a hierarchy types.

Other Approaches: Our design is inspired by prior event-based systems such as mOS [12] and Bro [22], but there are three main differences. While systems such as mOS allow user defined events, they perform all event processing in a single monolithic process. In contrast, we separate stacks and NFs, and allow events to traverse process boundaries via well-defined interfaces. This provides better performance isolation compared to running all threads in a single process and makes it possible to assign different resources and security levels for stacks and NFs. Our approach also allows NFs to be packaged as container-based appliances, which increases ease of deployment compared to tracking all dependencies of shared libraries in a multi-threaded process. Second, Microboxes's structured event types facilitate NF composition with inheritance allowing NFs to extend the base types while maintaining interoperability. Correspondingly, our event types focus on the data structures of message formats, as opposed to the definition of signaling rules that trigger an event (e.g., the mOS UDE filter functions). Finally, Microboxes's controller can use the publish/subscribe types in an NF definition to validate which NFs can be connected together, which is valuable when deploying loosely connected NFs developed by different organizations. In comparison, prior schemes require the linking to be done by individual NF developers.

3.2 μStacks

The simplest μEvents in the Microboxes Architecture are published when the packet first receives Layer 2/3 processing. However, NF development can be simplified by allowing subscription to more complex events that are the result of higher level protocol stack processing, such as reconstructing new data in a TCP bytestream or detecting the completion of the 3-way handshake. Microboxes extracts protocol processing from the NFs and implements it as modular μStacks that can be broken into different layers depending on NF needs.

Figure 3(c) shows our five μStacks and how they can be used by different NFs to meet their processing needs. In this example, only the Cache function requires full TCP endpoint termination, while the IDS needs no TCP processing at all. Despite their modularity, the μStacks all must build upon each other to prevent redundant work. We describe the full range of stack modules in Section 5.

Our μStack implementations build on mOS [12], which provides an efficient networking stack for monolithic middleboxes and its twin stack design enables tracking L4 states of both end-points. This allows NFs to subscribe to events related to the client or server side of a TCP connection. However, simply using mOS as a μStack is not sufficient due to consistency issues and performance challenges, which we will describe in Section 4.

The TCP μStacks each provide additional functionality, but modify data in a shared flow table. Events must be tied back to the original packet that produced them so that the stack knows when application layer processing has completed. Once all NFs finish with events, a message is returned to the associated μStack so that it can finalize processing. This is done for each μStack layer, eventually returning to the L2/L3 stack, which will propagate the packet out the NIC.

3.3 Microboxes Controller

The Microboxes Controller has two main roles. First, it maintains a registry of NFs and the event types that are being published or subscribed. Second, it acts similar to an SDN controller, containing the high level logic specifying how publishers and subscribers are interconnected. This architecture exploits similar ideas that are used in Microservices [10] and SDNs—NF functionality is broken down into small, easily replicated components that interact with simple event-based messaging APIs, and a centralized controller connects the components together to achieve the overall platform goals.

Each network function or μStack module registers a set of input ports (subscriptions) and output ports (publications) with the controller, along with the message type for each port. The Microboxes Controller is responsible for linking the input/output ports of NFs and performing type checking to ensure NFs will be able to interpret incoming event messages. This is achieved using the hierarchy of μEvent Types.

We assume that each event type is defined with a unique identifier and data structure known to other NF developers. For exposition we use names such as `PKT/TCP/SYN`, but in practice these are converted to unique identifier bits for each level in the hierarchy. NFs and μStacks announce what events they will publish at init time by requesting output ports from controller using mb_pub API. Meanwhile, they announce subscribed types to controller using mb_sub API. For every NF subscription, the Controller maps it to one or more publication ports. The Controller performs type checking to ensure the event types are the same, or the publication is a descendant of the subscribed type. NFs also implicitly publish all event types to which they subscribe; this can be used by the Controller to form chains of NFs. If an NF performs modifications to packets or stack state as part of its event processing, it specifies this with its publication, which allows the Controller to determine which NFs can run in parallel or in sequence, as described later.

3.4 Microboxes Applications

A Microboxes Application is composed of one or more μStack modules and multiple NFs. Several Applications can co-exist, and NFs only use the stack modules they require. This architecture allows NF developers to write generic modules, which can then be flexibly combined to produce more complex services by the Controller. Here we present two examples of the flexibility this provides.

Our TCP Splicer μStack implements a partial TCP stack capable of redirecting a flow after the first payload data arrives, while relying on a separate load balancing NF to select the appropriate backend. An example is shown in Figure 3(c), where an HTTP Load Balancer NF provides the policy to control the TCP Splicer μStack. To achieve this, the TCP Splicer publishes a `FLOW_REQ` message, which indicates that the handshake is complete and contains a pointer to a reconstructed bytestream incoming from the client. The HTTP LB NF subscribes to this event, inspects the request payload, and publishes a `FLOW_DEST` message that is returned to the TCP Splicer. The Splicer can then initiate the 2nd leg of the TCP connection and begin forwarding packets.

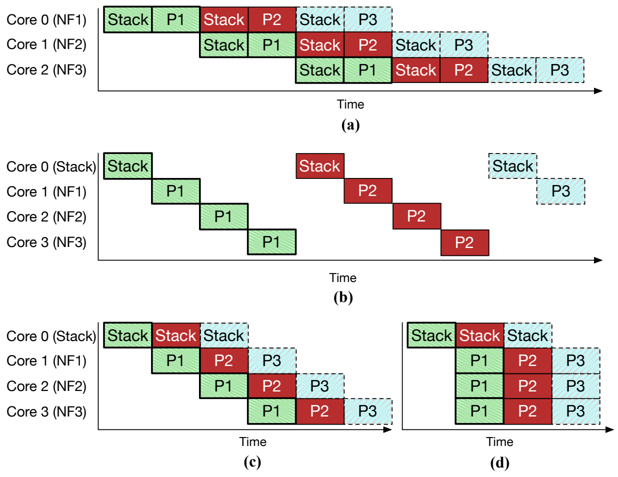


Figure 4: Stack pipelining and parallelism

The Splicer is not tightly coupled to the HTTP LB NF—the Microboxes Controller could, for example, connect the Splicer's `FLOW_REQ` publication to a Memcached LB NF that would apply a different load balancing policy before returning a `FLOW_DEST` message. By filtering how events are propagated from publishers to subscribers (e.g., based on destination port), the Controller can connect a single shared Splicer μStack to many different policy engines in order to redirect requests for a wide range of protocol types.

The event hierarchy also provides flexibility at the time of subscription. For example, a simple IDS NF might define an input port that subscribes to `PKT` events. Depending on the needs of the overall Microboxes Application, the controller might directly link this NF to the `PKT` output of the Layer 2/3 stack (as shown in Figure 3(c)), or it might connect it to the `PKT/TCP` output port of a TCP stack or subsequent NF. Both of these events match the subscribed type directly or via inheritance, so the IDS NF will be able to interpret the messages, ignoring the additional feeds included in the `PKT/TCP` variant.

4 ASYNCHRONOUS μSTACKS

In this section we first formalize the consistency challenges faced by performing NF and stack processing in parallel. We then describe the techniques used by Microboxes to overcome these issues.

4.1 Consistency Challenges

In a monolithic or pipelined system where each NF incorporates its own protocol stack, packets are processed to completion in one processing context, so there are no concurrency issues to consider. With Microboxes, execution is spread across multiple processes, and stack and NF processing can happen at the same time for packets in the same flow. We seek to provide identical ordering semantics that is achieved in a system that employs a protocol stack for each NF. This is shown in Figure 4(a), where three NFs sequentially process green, red, and blue packets; each NF is assigned its own CPU core, depicted in the distinct rows along the Y-axis; each row in the diagram represents the processing timeline of the NF on its own core. Once NF 1 performs protocol stack and its function processing on the green packet, P1, it passes the packet to NF 2 and begins processing the red packet, P2, and so on. While this architecture allows for pipelining across NFs, there clearly is wasted work from performing the stack processing repeatedly in each NF. However, naively moving the stack processing to its own core can cause consistency problems between the stack and NF processing. At a minimum, it adds unnecessary latency.

A safe, but slow approach is shown in Figure 4(b), where the stack is moved to a separate core from the NFs and the packets go through the 3 NFs on different cores in sequence. Here consistency is ensured across both NFs and the protocol stack by pipelining all dependent processing. Note that the NFs do not necessarily need to be idle during the blank spaces in the timeline—they could process packets from other flows since stack and NF state are assumed to be independent across flows. Thus, while this approach clearly hurts latency, it may not have a negative impact on throughput if a scheduler can effectively find packets from other flows to process while ensuring a consistent ordering.

Stack Consistency: Figure 4(c), shows a far more efficient timeline used by Microboxes, but this could lead to *stack inconsistency*. This happens when an NF attempts to read the protocol stack associated with a packet, but the stack state has already changed based on new packet arrivals. For example, when NF 2 starts processing the green packet P1, it might query the consolidated stack for protocol state such as the connection status; however, since the stack has already finished processing the red packet P2 at that point, the stack may have been updated to a state that is inconsistent with P1. It should be noted that while the diagram shows one phase of stack processing followed by a phase of NF processing for each packet, in our implementation NFs can respond to stack events related to the client-side or server-side stack processing. This causes further complex interleavings between the stack and the NF that Microboxes must handle to prevent inconsistencies.

NF Consistency: An even more desirable timeline is shown in Figure 4(d), which incorporates parallelism between the stack and NFs, as well as among the NFs themselves. To support this Microboxes requires not only techniques for ensuring consistency of the stack, but also preventing multiple NFs from manipulating packet data at the same time. For network functions operating at line rates, traditional concurrency techniques such as locks can cause unacceptable overheads, but several prior works have pointed out that NF parallelism can be achieved in cases where functions are primarily read-only or do not read/write the same packet data structures [28, 30].

While the preceding discussion has focused on packet processing and the stack, the same issues are relevant for Microboxes NFs processing events. The event messages that are published by the stack contain pointers to stack state and packet data, so consistency is a key concern.

4.2 Asynchronous, Parallel Processing

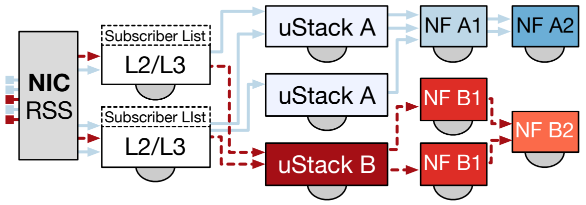
Microboxes employs four main techniques to ensure that NF and stack processing is efficient and consistent.

Asynchronous Stack Snapshots: are used to support parallelism of protocol stack and NF processing for packets within the same flow. Stack processing of a packet involves updates to both TCP connection initiator and responder state, e.g., sequence numbers, bytestream content, etc. Microboxes must guarantee that when an NF processes an event, it can obtain the stack state at the time the event was triggered. To achieve this, each event message contains a stack snapshot, i.e., a representation of the state at the time of the event and a timestamp. Some of this data, such as sequence numbers and connection state, can be copied into the event data structure. However, copying the bytestream for each event is infeasible. Fortunately, this can be avoided since the bytestream can be treated as an append-only data structure, thus the stack snapshot just needs to store a pointer to the end point of the bytestream at the time of the event. To ensure the bytestream behaves as append-only despite packet re-orderings, the μStack that is creating the bytestream maintains a "frontier" pointer indicating the furthest part of the stream that has been completely filled in. Events related to the bytestream are only released once the frontier is updated, and the event indicates that only data prior to that point is considered stable. The use of stack snapshots avoids the Stack Consistency problem described above, and allows stack and NF processing to be performed asynchronously, similar to Figure 4(c).

Parallel Events: are used to achieve full parallelism of both NF and stack processing (Figure 4(d)). The Controller can link several NFs' subscriptions to the output of a single NF acting as a "Splitter." In this case the event is produced with a reference count that is incremented for each NF that must process it. Once the NFs finish with the event, they respond to the splitter NF which tracks the reference count. When all NFs complete, the event can be propagated to the next stage as dictated by the controller. Having the Splitter NF handle the responses allows it to merge each NF's results in an NF-specific way, if necessary. For example, a Firewall NF might multicast an event to several different security monitoring

NFs in parallel, then based on the responses decide whether to block or allow the packet. As discussed further in Section 6, we assume that parallel NFs are read-only in nature, and thus will not conflict with each other when accessing packet or bytestream data associated with an event; NFs that perform writes are serialized. This assumption has been shown to be reasonable for many types of NFs in other studies [28, 30].

Modifying Packets and Stack State: Maintaining consistency of both packet and stack state is complicated when NFs perform writes. NFs that modify packet (or other event) data must specify this in their output port definition. The Controller can then use this to know whether NFs can be run in parallel. NFs are never allowed to directly update stack state, since that would cause race conditions between an NF and the μStack processing other packets in the flow. Instead, an event-based API is exposed to allow NFs to manipulate stack state. For example, the TCP μStack subscribes to `FLOW/TCP/TERMINATE` events, which might be published by a Firewall NF that wants to disallow a connection and send a `RST` packet. This avoids concurrency issues with stack state, but means that NFs must be able to handle asynchronicity, e.g., the Firewall NF may need to drop subsequent packets already processed by the stack prior to the event.



Parallel Stacks: Finally, to maximize performance in a multi-core environment it may be necessary to run multiple copies of the protocol stack on several cores. To support this, Microboxes uses Receive Side Scaling (RSS) support in the NIC with a bidirectional flow consistent hash. This ensures that all packets from a single flow will be sent to the same stacks and NFs, but allows for different flows to be uniformly distributed across cores. As shown above, the L2/L3 stack uses its subscriber table to determine the destination stack type (A or B) and then publishes packet events to the appropriate μStack instance, using the RSS value computed by the NIC to distribute across replicated stacks. NFs can also be replicated in a similar way (e.g., NF B-1). In addition to flows, data structures, such as TCP flow state table, are also partitioned across cores to avoid contention among parallel stacks.

5 CUSTOMIZING μSTACK MODULES

The type of TCP processing performed on each flow can be configured by adjusting the type of μStack modules that events are propagated through. Some stacks also allow fine grained configuration details to be adjusted on a per-flow basis to eliminate unnecessary processing. The μStacks build on each other to provide progressively more complex processing. Events from these stacks are then sent to NFs that perform the desired middlebox logic. The Microboxes TCP stack is broken into the following μStack modules:

Network Layer: When a packet arrives, it first receives Layer 2/3 processing to determine what flow it is associated with. This stack maintains minimal state–flow stats such as packet count and flow status tracking whether the flow is active or expired. It publishes `PKT` and `FLOW` events.

TCP Monitor: A TCP Monitoring stack seeks to track the TCP state at both the client and server side of a connection. This allows a monitor NF to reconstruct a bidirectional bytestream and observe TCP state transitions, for example as part of a stateful firewall, but does not allow full termination or arbitrary transformations to the bytestream. To support such TCP monitoring, we build upon the mOS TCP middlebox library, which divides stack processing between the client and server side for each packet [12]. The TCP Monitor subscribes to `PKT` from the Network Layer and produces events including `EVENT/DATA_RDY`, signifying a change in the bytestream, and `FLOW/TCP`, indicating a change to the TCP state. NFs that monitor these events can further narrow down their scope by specifying for each flow whether to subscribe to events for updates to the state of only the client, the server, or both. Finally, the processing overhead of the TCP Monitor can be tuned by specifying whether to reconstruct the TCP bytestream on a per-flow basis.

This type of stack is useful for middleboxes that observe TCP state changes or need bytestream reconstruction. NFs that subscribe to `EVENT/DATA_RDY` can make in-place modifications to the bytestream before it is forwarded to the client or server, but they cannot make arbitrary changes to the data since changing the size of packets would disrupt the TCP state (i.e., sequence numbers) at the client and server. Thus the TCP Monitor stack provides a lightweight TCP library for middleboxes that primarily observe flows at the transport layer or above.

TCP Splicer: A desirable operation for proxy-type middleboxes is the ability to redirect a TCP connection after the handshake has been established. For example, an HTTP proxy might observe the contents of a `GET` request before selecting a server and forwarding the request [9, 18]. The Microboxes TCP Splicer stack simplifies this operation by extending the TCP Monitor stack, without requiring the complexity of a full TCP endpoint. The Splicer μStack works by subscribing to the TCP Monitor's `PKT/TCP` event to detect when the client initiates a connection to an IP address configured with the Splicer. The Splicer then responds with a SYN-ACK packet so that it can process the three-way handshake with the client. The Splicer publishes a `FLOW_REQ` event once the handshake completes and data has arrived from the client. A user-defined NF hosting the proxy logic will listen for that event and respond with FLOW_DEST containing the new destination IP. Once the Splicer obtains this information, it initiates a new TCP handshake with the selected destination server. All subsequent packets in the flow can then go through a fast path in the Splicer that requires only simple modifications to the TCP header sequence numbers [18], *allowing zero-copy TCP splicing*.

While this could also be achieved by terminating the connection in the middlebox, like in a split proxy [16], that would require additional TCP logic, which incurs unnecessary overhead if the NF only needs to redirect the flow without modifying the bytestream.

TCP Endpoint: Microboxes NFs can also act as TCP endpoints, for example to host a cache that can respond directly to some requests. The TCP Endpoint μStack module further extends the TCP Monitor to contain full TCP logic (congestion control, retransmissions, etc). It then exports events similar to a socket API. Our implementation leverages mTCP [13], on which mOS is also based, which allows NFs to work with an EPOLL interface to the μStack's sockets. Microboxes NF developers can then work directly with the EPOLL interface popular for high performance socket programming.

Supporting TCP endpoints opens new opportunities in the types of services that can be deployed. For example, a Microboxes deployment at an edge cloud might provide transparent middlebox services for most flows, but directly terminate and respond to a subset of other flows, with functions such as CDN caches or for IoT data analysis.

TCP Split Proxy and Bytestream NFs: The most complex NF types perform transformations on the bytestreams traversing them; we call these Bytestream NFs. To do so requires two TCP connections, one with the client and one with the server. This allows both redirection (similar to the TCP Splicer stack), as well as arbitrary bytestream transformations. The Microboxes Proxy μStack module is implemented as two TCP Endpoint stacks. The Proxy μStack publishes a FLOW_REQ event when a new connection arrives. The Bytestream NF subscribes to this message and responds with a FLOW_DEST event to inform the Proxy how to redirect the flow. The DATA_RDY message type is used both by the Proxy μStack and the Bytestream NF to indicate when they have incoming or outgoing data ready, respectively.

6 MICROBOXES IMPLEMENTATION

Microboxes is built on the OpenNetVM [29] NFV framework, which provides shared-memory based communication for NFs running as separate processes or containers, and an IO management layer for starting/stopping NFs and sending/receiving packets over the NIC. Here we focus on the implementation of the Microboxes TCP stack and shared memory-based event communication system.

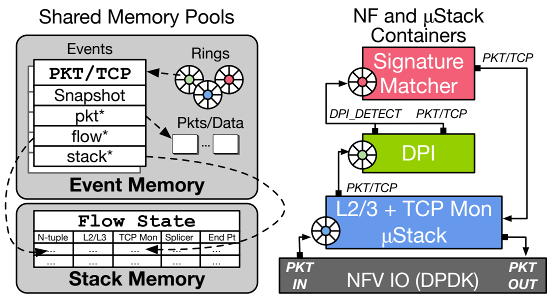


Figure 5: Microboxes architecture

Stack Modules: Our TCP μStack modules are based on mOS [12] and mTCP [13], with significant changes to support customization and a consolidated stack shared by many NFs. TCP processing in Microboxes is based on the mOS monitor stack, with additional code from mTCP to support full endpoint termination. We modified 5.8K lines of C code in mOS and have an additional 13.5K lines for our NFV management layer and sample NFs. Our stack incorporates two key differences from these platforms: 1) we streamline TCP processing and separate it into our μStack modules to allow for a modular deployment, and 2) we decouple the stack from the NF library and use dynamic subscriptions to control how events are delivered.

mOS and mTCP are designed to be fully-functional, compatible TCP stacks, whereas Microboxes seeks to provide a range of stack types depending on NF needs. Thus the minimalist Microboxes TCP Monitor μStack does not maintain, for example, ACK/retransmission queues, resulting in a smaller memory footprint and higher performance. For stack modules that require termination we use mTCP, with integration into our platform so that packets arrive as events from the lower-layer stacks instead of directly accessing the NIC. We also implement the Splicer μStack module which extends the TCP Monitor to add redirection capabilities.

From our observation that the TCP stack can often be the bottleneck, we move unnecessary processing out of the stack and into the NF, for example we extract the event processing code in mOS to a separate library which is incorporated into the NFs themselves to perform flow-level event filtering. Each stack module can be run in its own process, although we combine the Layer 2/3 and TCP Monitor stacks into one since L2/3 processing adds minimal extra cost. This separates the stack and NFs into different protection and performance domains, providing greater isolation.

Together, these changes allow us to tune the TCP processing performed on a per-flow basis, and ensures that stack processing is made as lightweight as possible.

Middlebox App	μStack	Description
IDS (DPI+Sig_Match)	Monitor	First a DPI NF uses the nDPI [3] open source library to detect application protocols. Then the Signature_Match NF examines a set of inspection rules to match attack signatures.
Monitor (Flow_Stats+Logger)	Monitor	A Flow_Stats NF publishes statistics events for network connections. A Logger NF records the timestamp and values of each event it receives.
Layer 4 LB	Layer 2/3	This NF uses a 5-tuple hash to split traffic and rewrite the packet's destination port and address. It does not require TCP stack processing.
Layer 7 LB	Splicer	The HTTP_LB NF uses the HTTP GET request to choose a destination server and balance the load. The Splicer μStack maintains the connection with the client side and initiates the connection to the selected backend.
Proxy	Split Proxy	Establishes separate sockets to both client and server, then uses EPOLL to check for data in events from either socket, and pushes that data from one socket to another.
Lighttpd	Endpoint	This legacy web server can serve static files (jpg/html/etc) using the TCP Endpoint μStack.

Table 2: Sample Microboxes Applications using different μStack modules.

Shared Memory and TCP State: Microboxes sets up two shared memory pools: Event Memory and Stack Memory as shown in Figure 5. The first is used to store packets, events, reassembled flow data, and communication queues which support zero-copy data movement between NFs and μStacks. The Stack Memory pool stores the flow tables used to maintain per-flow TCP state, including information such as sequence numbers and connection status. This data is stored in a separate pool and can only be accessed via a restrictive API to keep all state data read-only for NFs.[1]

Microboxes includes an *optimized read path* that eliminates all modifications to TCP state to improve concurrency. First, this is achieved by maintaining TCP state in the shared Stack Memory, allowing zero-copy access to much of the data for NFs (with the exception of data copied into stack snapshots to ensure consistency). Second, we avoid indirections caused by hash table lookups as much as possible by providing direct pointers to state table entries and the reassembled bytestream with event messages. This can have a substantial impact on services operating at line rates. Third, operations to read TCP state are all stateless functions, e.g., when querying the bytestream to retrieve data, the API does not track the position of the last read; instead the API is stateless and requires the NF to specify the desired offset. This is in contrast to the original mOS/mTCP APIs, which updated internal state with some operations. For example, `mtcp_peek` updates the offset of already retrieved data, and `mtcp_ppeek` has an on-demand memory allocation, causing these operations to update stack state and requiring them to be performed sequentially. By making these APIs stateless, Microboxes has fewer cache invalidations and concurrency issues to deal with than mOS when extended to support NF chains.

Microboxes divides flows using RSS (Receive Side Scaling) so that each TCP μStack instance maintains its own set of flows in partitioned memory. This prevents any concurrent updates to shared cache lines from different stack instances, which would cause cache coherence traffic.

Stack State Snapshots: Microboxes embeds Stack Snapshots in TCP event messages so that the TCP stack can continue updating flow table state without inconsistencies from concurrent updates. The stack snapshot only needs to contain data from the state table that could be potentially modified when the stack processes subsequent packets in the flow, but is not accessible in the packet that is referenced by the event. For example, the latest sequence number of the sender can be trivially read from the packet itself (referenced by the event), but the latest sequence number of the receiver needs to be stored in the stack snapshot since it is not available in packet data and may be updated if a return packet arrives. Similarly, the current connection status (e.g., awaiting SYN-ACK, connected, etc.) is not found in the packet and could be updated, so it also must be included in the snapshot. An offset pointer for the reconstructed bytestream is provided in the snapshot, allowing the stack to append data from subsequent packets of the same flow without affecting consistency. In total, the stack snapshot is only 23 bytes, which is substantially smaller than the full state. The time for making the snapshot for each packet is around 76 cycles, even for a large number of flows.

Event Messages: For each event, a single message data structure is allocated in the shared Event Memory pool by its publisher. The producing NF then checks its subscription list and adds a pointer to the message data structure for the event queues of all subscribed NFs. As shown in Figure 5, each NF has a single incoming queue, but this can be linked to multiple output ports from one or more other NFs/μStacks. Event Messages are designed to be compact, so they only include metadata such as the event type and snapshot. For packet data or for state that may need to be modified by NFs, the structure contains data pointers that can be used to reference larger amounts of data, e.g., the packet that triggered the `PKT` event or the TCP Monitor's flow state. The Microboxes "run loop" on each NF polls for incoming events and triggers a handler function that the NF specifies for each event type.

[1]Microboxes assumes that NFs are not malicious and will not try to corrupt TCP state or snoop on other flows' data by using arbitrary memory operations; we believe this is an acceptable assumption for network provider operated NFV platforms where NFs are thoroughly vetted before deployment. For less secure, multi-tenant environments, the state memory pool could be kept in read-only memory with finer grained access controls.

```
1  void startIDS() {
2    mb_nf_mem_init(cntrlr); // map shared memory
3    // announce subscribed types to controller
4    mb_sub(cntrlr, PKT_TCP, cb_noop);
5    mb_sub(cntrlr, DPI_DETECT, cb_detect);
6    // request an output port for publication type
7    port = mb_pub(cntrlr, EVENT_IDS_ALERT);
8    mb_nf_run(); // enter run loop
9  }
10 void cb_detect(DPI_DETECT_msg m) {
11   if(m.dpi_app_type == HTTP)
12     mb_up_sub(cntrlr, PKT_TCP, e.flow, cb_http);
13   else if(m.dpi_app_type == SQL)
14     mb_up_sub(cntrlr, PKT_TCP, e.flow, cb_sql);
15 }
16 void cb_sql(PKT_TCP_msg m) {
17   if(sql_injection_attack(m.payload)) {
18     IDS_ALERT_msg m2 = mb_new_msg(IDS_ALERT)
19     // fill in m2
20     mb_publish_event(port, m2)
21   }
22 }
```

Listing 1: IDS Signature Matching NF

Since all events arrive in a single queue, NFs process them in arrival order regardless of event type. Our μStack defines its TCP events using the mOS base events as building blocks, so it inherits the same ordering enforced by mOS, and our message communication mechanism guarantees the events are processed in order for NFs within a chain. However, we don't provide ordering for NF internal events and we assume this is handled by the developers.

Sample Applications: We have ported a set of middlebox applications into Microboxes using our event and stack APIs. Where possible, we have decomposed the applications into separate NF modules and offloaded processing to the stack rather than inside the NF. Table 2 lists out these applications and the type of μStack they make use of.

These applications provide a range of examples with different protocol processing needs. Most applications have been decomposed into multiple NF modules to further stress our event system. In addition to writing our own NFs, we also include the Lighthttp web server to illustrate support for legacy applications. The web server uses our TCP endpoint μStack. The Controller can also link together several of these applications, for example to provide a Layer 7 HTTP load balancer in front of the Lighttpd server. These deployments involve multiple μStacks that Microboxes will automatically keep consistent.

In Code Listing 1, we present simplified code for our IDS Signature Matching NF (the full version performs more complex analysis based on the pattern matching code of Snort). The `startIDS()` function initializes subscriptions, setting a default "no op" handler for incoming TCP packets and the `cb_detect()` callback for events arriving from the DPI NF that detects application-layer protocols (lines 4-5). Based on the DPI detection event's type, the NF updates the subscription to the appropriate callback function to perform signature matching on subsequent packets that arrive in the flow (lines 11-14). If an attack is detected, a new `IDS_ALERT` message is published (line 20).

7 EVALUATION

Testbed Setup: Our experiments were performed on the NSF CloudLab testbed [24] with "c220g2" type servers from the Wisconsin site. Each server has Dual Intel Xeon E5-2660 v3 @ 2.60GHz CPUs (2*10 cores), a Dual-port Intel X520 10Gb NIC and 160GB memory. All servers run Ubuntu 14.04 with kernel 3.13.0-117-generic and use Intel's DPDK v16.11. We use two sets of traffic generators, mTCP-based [13] web client and server for high speed tests, and Nginx 1.4.6 and Apache Bench 2.3 for setting up multiple servers and splitting up web traffic.

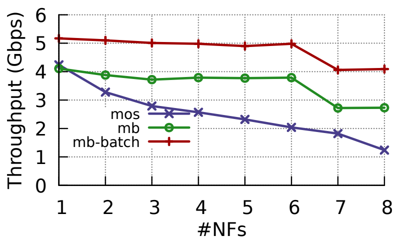
7.1 Protocol Stack Performance

Stack Consolidation: To evaluate the benefit of consolidating the stack for chained NFs, we first port the mOS middlebox library to our NFV platform. mOS by itself does not support chaining, and assumes a single NF is used with complete control over the NIC. A straightforward port of chained mOS NFs (labeled "mos" in figures) uses a packet interface between instances and performs stack processing for each NF. In contrast, our Microboxes system (labeled "mb" in figures) only needs to perform stack processing once, and then forwards the stack events through the chain of NFs. For both cases, the NFs themselves perform negligible processing and we use a workload of 4000 concurrent web clients accessing files 64B or 8KB long.

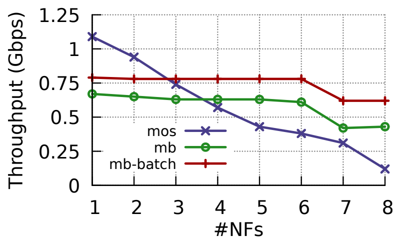
The results in Figure 6a and Figure 6b, show that as the chain length increases, the throughput of Microboxes remains mostly flat, while mOS drops quickly. For a chain of 6 NFs, Microboxes outperforms mOS by up to 105% for requests for a 64B file and 144% for 8KB file (including batching). Batching events at each NF contributes around 30% of this improvement (compare mb-batch vs. mb); we use batching for the remaining experiments. Microboxes does see a throughput decrease when the chain is longer than 6 since it has to use cores on another socket and pay NUMA penalties.

As shown in Figure 6c, Microboxes also reduces latency by 20% due to stack consolidation with longer NF chains. mOS gets marginally better latency only when there is one NF since it runs the stack and NF on the same core, while Microboxes runs the stack on a separate core and cross-core communication impacts the performance for this case.

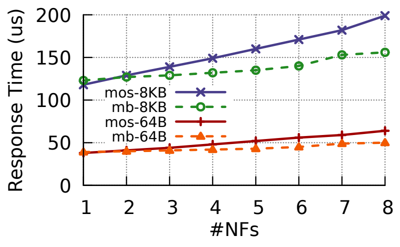
Parallel Processing: To evaluate the performance of parallel processing, we compose multiple DPI NFs together in a sequential or parallel manner. The first instance is set up



(a) 8KB Reqs

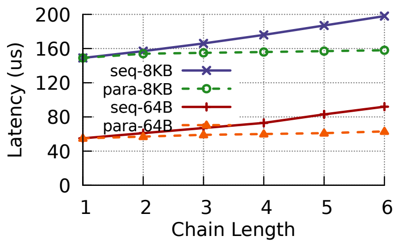


(b) 64B Reqs

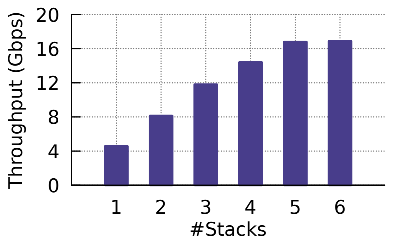


(c) Latency

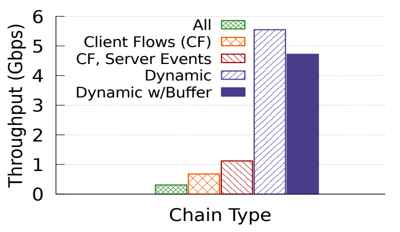
Figure 6: Microboxes improves throughput and latency by eliminating redundant stack processing in chains.



(a) Parallel NFs



(b) Multi-Stack



(c) DPI-Chain

Figure 7: (a) Parallel processing performance (b) Multistack performance (c) Event subscriptions impact

as the head of the chain and controls the parallelism for the following instances. The throughput for parallel and sequential chains are similar (not shown). We can see the latency difference from Figure 7a. As the number of NFs increases, the latency for sequential chains increases linearly, while the parallel chain is almost flat. For 6 NFs, parallel processing can reduce latency 32% for a 64B web request and by 20% for 8KB web requests.

Scalability: We next evaluate the scalability of our μStack when using multiple cores and two dual port NICs. We use two pairs of clients and servers to generate the web traffic (8KB HTTP file requests) through Microboxes. At each stage of the experiment, we add one more core to host the stack and use the RSS value to split flows across the stacks. From Figure 7b, we can see a linear speedup for total processing rate until the Ethernet link (2*10Gbps) becomes the bottleneck.

These experiments show that our architecture can efficiently run chains of NFs with improved throughput and latency characteristics due to elimination of redundant work and better parallelism. We next show how customizing the stack for different NFs can provide further performance improvements while maintaining deployment flexibility.

7.2 Load Balancer: Flexibility and Speed

In this experiment, we show how our layered TCP stack can provide different performance and functional requirements to meet an NF's needs. We use Nginx as a web server and Apache Bench as the client to generate web traffic. We consider four different Microboxes NFs to load balance the traffic: L4_LB, L7_LB and Bytestream Proxy, based on our Monitor, Splicer and Split Proxy Stacks respectively. We compare against a baseline of the simple DPDK L2 forwarder example NF and HAProxy [2], a popular open source load balancer which uses Linux kernel-based networking. We also evaluate a "L7 LB + Cache" approach which combines our HTTP load balancer with the Lighttpd server running on the TCP Endpoint μStack. We consider the case where 50% or 100% of the requests are redirected by the Splicer to the local Lighttpd server that acts as a faster cache compared to Nginx.

From Table 3, we can see L4_LB has the lowest latency and highest throughput of the Microboxes solutions. This is as expected since it only looks at header information, and does minimal TCP processing (e.g., no bytestream reconstruction). This is a valid choice for simple load balancing scenarios where redirection does not need to be based on the content of the request. The L7_LB adds more overhead since it needs to redirect the connection and look at application data to select a server. However, it also provides greater flexibility since the destination server can be determined after the HTTP GET request has been received. The range of applications supported by our system is best demonstrated by the L7 LB + Cache test since it shows the value of being able to deploy both middleboxes and end server applications on an integrated platform. Using Lighttpd as a cache can increase throughput by 29% or more compared to directing requests to Nginx via the L2 forwarder. The Microboxes Bytestream Proxy has the largest overheads as it needs to maintain connections for both server side and client side. With this cost,

NF Type	Latency(us)	Reqs/s	Norm.
DPDK L2 Fwd	189	30,174	1
HAProxy	375	18,356	0.61
L4 LB	199	28,894	0.96
L7 LB	210	27,772	0.92
L7 LB + 50% Cache	129	38,981	1.29
L7 LB + 100% Cache	90	41,372	1.37
Bytestream Proxy	255	23,984	0.79

Table 3: The Bytestream Proxy, L4 and L7 load balancers use different μStacks to provide trade-offs in load balancer flexibility and performance; all can outperform HAProxy, and integrating a cache with the LB can increase throughput by 1.29X or more.

however, comes the opportunity to transform the bytestream in arbitrary ways. HAProxy also uses two sockets, yet it has substantially higher latency and lower throughput than the other approaches since it is not based on an optimized stack or an NFV IO platform.

This experiment demonstrates the customization that Microboxes can offer—depending on the needs of a specific load balancer it can select the appropriate stack (minimalist TCP, Splicer, or proxy stack). Compared to a traditional approach like HAProxy, which is tightly coupled to a full TCP stack implemented in the kernel, this can provide between 31% to 57% improvement in throughput and a 32% to 47% reduction in latency while only using one core. Extending this further to use the endpoint stack to host a legacy web server as a cache provides further benefits. To our knowledge, Microboxes is the first NFV platform to support a unified deployment of middleboxes and servers.

7.3 IDS: Customizing Subscriptions

In this experiment, we explore how the ability to filter the stack events an NF subscribes to affects performance. We chain two NF modules, DPI and Signature_Match, together to work as an IDS middlebox. Events from the TCP Monitor μStack go through DPI first and then Signature_Match. The throughput of this chain is dominated by the Signature_Match module as it is much slower than DPI. We use five different configurations to show how stack customization and dynamic subscriptions can affect the performance.

For the first configuration, both modules register for the `PKT_IN` event of the TCP Monitor Stack, causing events about all packets to go through both NFs. This configuration gets the lowest throughput since both NFs receive and process two events for each packet: one when the stack updates the client-side state and one when it updates the server-side state. For the second configuration, we deregister events for server side flows and only allow events related to the client's packets to pass through, effectively reducing the number of events by about half. For the third configuration, we deregister the events generated by the client stack and only observe events related to server stack updates caused by packets from the client, and this reduces another half of events. For most IDS deployments, this would be sufficient: the IDS only needs to monitor traffic entering from untrusted clients, and it does not need to separately track TCP state for both the client and server sides. From Figure 7c, we can see 1.2x and 2.6x throughput improvements shown in the second and third bar due to narrowing the subscribed events when compared with the first configuration.

Next we enable dynamic subscriptions. For the fourth configuration, DPI subscribes to `PKT/TCP` events for the server stack, as in case 3 above. However, we now have the DPI publish events based on the type of protocol identified, and the Signature_Match NF subscribes for `EVENT/DPI_DETECT` messages. Thus when DPI identifies a flow as HTTP, it triggers the event and then uses dynamic subscriptions to unsubscribe from `PKT/TCP` events for that flow (since it has already been identified). At the same time, the Signature_Match NF begins to subscribe to the `PKT/TCP` events so that it can perform its analysis (e.g., scanning for SQL injection attacks). If a specific signature is found, Signature_Match will publish a new `EVENT/IDS_ALERT` event and deregister the current flow. By using dynamic filtering, we reduce the number of processed events by more than 50%, and provide nearly 5x throughput increase compared to the static filtering in case 3. For the last configuration, we enable flow reassembly so that DPI and Signature_Match can subscribe and publish `DATA_RDY` events. This allows the NFs to detect patterns that span multiple packets, at the expense of about 15% overhead to reassemble the packet payload and create the bytestream.

This experiment shows the value of enabling NFs to tune their subscriptions and to be able to use events to communicate information. Prior systems would send far more packets to each NF for processing (similar to cases 1-3), while our approach allows NFs to be directly alerted of information they need and tune their subscriptions accordingly.

7.4 IDS+Mon: Dynamic, Parallel Events

We now evaluate a more complex group of NFs and show how dynamic subscriptions and parallel processing can improve performance under different configurations. Our setup is based on the real world services evaluated in NFP [28].

We first configure the five NFs as a sequential or parallel chain as shown at the top of Figure 8. In the "Default" configuration, all NFs in the chain subscribe to `PKT/TCP` events; however, since the Signature_Match and Logger modules are much more expensive than the others they are the performance bottlenecks. To tune the chain, we next

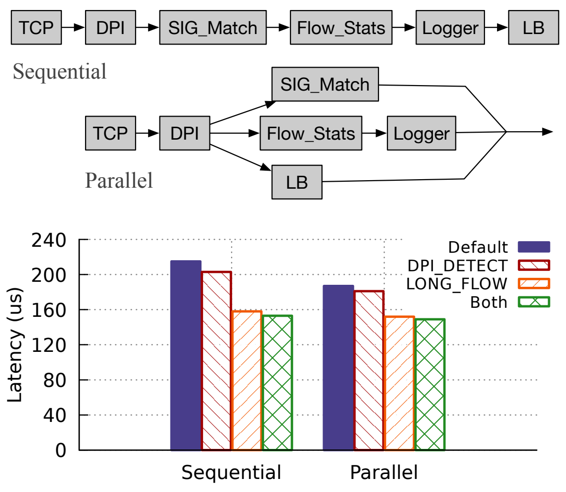


Figure 8: Parallelizing a long chain and using dynamic event filtering provides substantial latency benefits

setup the `DPI_DETECT` event described in the prior section which causes the DPI and Signature_Match NFs to dynamically tune their subscriptions. This reduces the load on the first two NFs, improving the average latency as shown by the "DPI_DETECT" bars in Figure 8. Next we consider custom events for the Flow_Stats and Logger NFs. We define a `LONG_FLOW` event published by Flow_Stats once the average packet length of a flow is larger than a threshold. The Logger subscribes to this event, and adjusts its subscription so that it only logs events about long running flows. Finally, we consider the combination of both of these custom events, which provides a further improvement in performance. Compared to the sequential chain, the parallel configuration provides between a 3%-13% latency reduction.

8 RELATED WORK

TCP Stacks: There are several research efforts on high performance networking stacks [5, 13, 15, 20, 21, 23] but most are specific to end-host applications. We present a comparison of NFV frameworks [6, 12, 22, 26] that support TCP protocol processing in Table 4. mOS [12] provides a reusable TCP stack to facilitate L4-L7 NF development for monolithic architecture, however it focuses on a single NF, not a chain. Simply replacing each NF's stack with mOS and hooking them together will incur multiple copies and cause performance issues. Comb [26] could support both pipelined and monolithic applications and has focused on consolidating applications and managing resources at the network-wide, without addressing the stack or NF configurability at the application level. Bro [22] presents a stack and event management framework specific to Intrusion Detection System. However its modules are tightly-coupled which makes it hard to extend into other applications. FlowOS [6] presents a flow-level programming model for middleboxes. It hides the low-level packet process but it relies on Linux kernel stack which is not optimized for middleboxes.

Framework	Architecture	Flexibility	Events	Chain
Microboxes	pipelined	high	yes	yes
mOS [12]	monolithic	high	yes	no
Comb [26]	pipelined /monolithic	low	no	yes
Bro [22]	monolithic	medium	yes	no
FlowOS [6]	monolithic	medium	no	yes

Table 4: A comparison with other middlebox frameworks that support TCP protocol processing.

Microboxes provides a shared customizable TCP stack for each group of NFs which is designed to remove redundant stack processing while maintaining consistency requirements. Our event system is inspired by the user-defined events in mOS and Bro, but we extend these with a meaningful type hierarchy and support for cross NF coordination.

Modularity and Parallelism: Click has been a popular platform to build network functions due to its modularity and extensibility. A set of individual elements are connected via pull or push connections to build a more complex application. Click also inspired a series of other modular NFV architectures [4, 8, 17, 26]. However, without native TCP support in Click, these platforms are restricted to L2 or L3 NFs. NFP and Parabox [28, 30] focus on NF level parallelism, they can automatically analyze NF dependencies and reconstruct a service graph with parallel NFs to improve performance. P4 [7] is a configuration language for packet processors and it enables parallelism by identifying table dependencies.

Microboxes works along both directions and is complimentary to these works. By decomposing a stack into several customizable building blocks, Microboxes promotes parallelism. We mainly focus on asynchronous processing that allows parallelism between NFs and the protocol stack.

9 CONCLUSIONS

Existing NFV frameworks focus on efficient packet movement and layer 2/3 processing, yet many key middlebox applications require higher level protocol processing. The Microboxes architecture balances the goals of consolidating protocol processing and supporting a customizable stack that can be tailored to the needs of individual flows. We achieve this by designing a modular, asynchronous TCP stack and efficient, event-based communication mechanisms to link together NFs into complex applications. We introduce optimizations such as stack snapshots to ensure stack consistency while maintaining high performance. We believe that Microboxes will provide a valuable framework for NF developers to deploy transport-layer-and-above middleboxes or end-services.

Acknowledgements: We would like to thank our shepherd KyoungSoo Park, the anonymous reviewers, and Jean Tourrilhes and Puneet Sharma from HPE for their detailed comments and valuable feedback. This work was supported in part by NSF grants CNS-1422362 and CNS-1522546.

REFERENCES

[1] Data plane development kit (dpdk). http://www.dpdk.org/.

[2] Haproxy. http://haproxy.1wt.eu/.

[3] ndpi | ntop. http://www.ntop.org/products/ndpi/.

[4] B. Anwer, T. Benson, N. Feamster, and D. Levin. Programming slick network functions. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR '15, Santa Clara, California, USA, June 17-18, 2015*, 2015.

[5] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, 2014.

[6] M. Bezahaf, A. Alim, and L. Mathy. Flowos: A flow-based platform for middleboxes. In *Proceedings of the 2013 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMiddlebox '13, pages 19–24, New York, NY, USA, 2013. ACM.

[7] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[8] A. Bremler-Barr, Y. Harchol, and D. Hay. Openbox: A software-defined framework for developing, deploying, and managing network functions. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, 2016.

[9] A. Cohen, S. Rangarajan, and H. Slye. On the Performance of TCP Splicing for URL-aware Redirection. In *Proceedings of the 2Nd Conference on USENIX Symposium on Internet Technologies and Systems - Volume 2*, USITS'99, pages 11–11, Berkeley, CA, USA, 1999. USENIX Association.

[10] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzar, F. Montesi, R. Mustafin, and L. Safina. Microservices: Yesterday, today, and tomorrow. *Present and Ulterior Software Engineering*, 2017.

[11] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. Softnic: A software nic to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[12] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park. mos: A reusable networking stack for flow monitoring middleboxes. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, 2017.

[13] E. Jeong, S. Woo, M. A. 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 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, 2014.

[14] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[15] R. Laufer, M. Gallo, D. Perino, and A. Nandugudi. Climb: Enabling network function composition with click middleboxes. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMIddlebox '16, New York, NY, USA, 2016. ACM.

[16] F. Le, E. Nahum, V. Pappas, M. Touma, and D. Verma. Experiences Deploying a Transparent Split TCP Middlebox and the Implications for NFV. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMiddlebox '15, pages 31–36, New York, NY, USA, 2015. ACM.

[17] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, and P. Cheng. Clicknp: Highly flexible and high-performance network processing with reconfigurable hardware. In *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, 2016.

[18] D. A. Maltz and P. Bhagwat. MSOCKS: an architecture for transport layer mobility. In *IEEE INFOCOM '98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings*, volume 3, pages 1037–1045 vol.3, Mar. 1998.

[19] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, et al. ClickOS and the art of network function virtualization. In *USENIX NSDI*, 2014.

[20] Z. Niu, H. Xu, D. Han, P. Cheng, Y. Xiong, G. Chen, and K. Winstein. Network stack as a service in the cloud. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 65–71, New York, NY, USA, 2017. ACM.

[21] S. Pathak and V. S. Pai. Modnet: A modular approach to network stack extension. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, 2015.

[22] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.

[23] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, Berkeley, CA, USA, 2014. USENIX Association.

[24] R. Ricci, E. Eide, and C. Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[25] L. Rizzo. netmap: A novel framework for fast packet i/o. In *USENIX ATC*, 2012.

[26] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, 2012.

[27] 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 *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 13–24, New York, NY, USA, 2012. ACM.

[28] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu. NFP: enabling network function parallelism in NFV. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, 2017.

[29] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, and T. Wood. OpenNetVM: A platform for high performance network service chains. In *HotMiddlebox*. ACM, 2016.

[30] Y. Zhang, B. Anwer, V. Gopalakrishnan, B. Han, J. Reich, A. Shaikh, and Z. Zhang. Parabox: Exploiting parallelism for virtual network functions in service chaining. In *Proceedings of the Symposium on SDN Research, SOSR 2017, Santa Clara, CA, USA, April 3-4, 2017*, 2017.