A Verified Session Protocol for Dynamic Service Chaining

Pamela Zave, *Fellow, ACM*, Fabrício B. Carvalho, Ronaldo A. Ferreira, Jennifer Rexford, *Fellow, IEEE, ACM*, Masaharu Morimoto, and Xuan Kelvin Zou

***Abstract*— Middleboxes are crucial for improving network security and performance, but only if the right traffic goes through the right middleboxes at the right time. Existing traffic-steering techniques rely on a central controller to install *fine-grained forwarding rules* in network elements—at the expense of a large number of rules, a central point of failure, challenges in ensuring all packets of a session traverse the same middleboxes, and difficulties with middleboxes that modify the "five tuple." We argue that a *session-level protocol* is a fundamentally better approach to traffic steering, while naturally supporting host mobility and multihoming in an integrated fashion. In addition, a session-level protocol can enable new capabilities like *dynamic* service chaining, where the sequence of middleboxes can change during the life of a session, e.g., to remove a load-balancer that is no longer needed, replace a middlebox undergoing maintenance, or add a packet scrubber when traffic looks suspicious. Our Dysco protocol steers the packets of a TCP session through a service chain, and can dynamically reconfigure the chain for an ongoing session. Dysco requires no changes to end-host and middlebox applications, host TCP stacks, or IP routing. Dysco's distributed reconfiguration protocol handles the removal of proxies that terminate TCP connections, middleboxes that change the size of a byte stream, and concurrent requests to reconfigure different parts of a chain. Through formal verification using Spin and experiments with our prototype, we show that Dysco is provably correct, highly scalable, and able to reconfigure service chains across a range of middleboxes.**

***Index Terms*— Session protocol, NFV, middleboxes.**

I. Introduction

IN THE early days of the Internet, end-hosts were stationary devices, each with a single network interface, communicating directly with other such devices. Now most end-hosts are mobile, many are multihomed, and traffic traverses chains of middleboxes such as firewalls, network address translators, and load balancers. In this paper, we argue that the "new normal" of middleboxes warrants a re-examination of approaches, as has happened with mobility [1].

Manuscript received November 13, 2019; revised July 20, 2020; accepted November 1, 2020; approved by IEEE/ACM Transactions on Networking Editor D. Malone. Date of publication November 19, 2020; date of current version February 17, 2021. This work was supported in part by the NSF under Grant CNS-116112, in part by the Brazilian National Council for Scientific and Technological Development (CNPq) under Grant 201983/2014-1, in part by the Brazilian National Research and Educational Network (RNP) under Grant 2956, and in part by the INCT InterSCity. *(Corresponding author: Ronaldo A. Ferreira.)*

Pamela Zave and Jennifer Rexford are with the Department of Computer Science, Princeton University, Princeton, NJ 08540 USA.

Fabrício B. Carvalho is with the College of Computing, UFMS, Campo Grande 79070-900, Brazil, and also with UFMT, Cuiabá 78060-900, Brazil.

Ronaldo A. Ferreira is with the College of Computing, UFMS, Campo Grande 79070-900, Brazil (e-mail: raf@facom.ufms.br).

Masaharu Morimoto is with NEC, Kanagawa 211-8666, Japan.

Xuan Kelvin Zou is with ByteDance, Shanghai 201600, China.

Digital Object Identifier 10.1109/TNET.2020.3037049

Most existing research proposals for middlebox insertion or "service chaining" use a logically centralized controller to install fine-grained forwarding rules in network elements, to steer traffic through the right sequence of middleboxes [2]–[9]. The many weaknesses of these solutions are a direct result of their reliance on forwarding rules for traffic steering:

- They rely on real-time response from the central controller to handle frequent events, including link failures, traffic fluctuations, and the addition of new middlebox instances.
- They need network state that grows with the number of policies, the difficulty of classifying traffic, the length of service chains, and the number of instances per middlebox type.
- Updates to rules due to changes in policy, topology, or load may change the paths of ongoing sessions, yet all packets of a session must traverse the same middleboxes ("session affinity").
- Fine-grained routing is inherently intra-domain. It is difficult to outsource middleboxes to the cloud [10] or other third-party providers [11], since the controller cannot control the entire path.
- Some middleboxes modify the "five-tuple" of packets in unpredictable ways, so that forwarding rules matching packets going into the middlebox might not match them on the way out.
- Some middleboxes classify packets to choose which middlebox should come next. These middleboxes should be able to select the service chain for their outgoing packets, which forwarding by network elements does not allow them to do.
- Adding middleboxes to a secure session (e.g., TLS) is challenging without cooperation with the end-hosts to exchange the information needed to decrypt and reencrypt the data [12].
- A multihomed host spreads traffic over multiple administrative domains (e.g., enterprise WiFi and commercial cellular network), yet some middleboxes need to see all the data in a TCP session (e.g., for parental controls [13]). In the administrative domain where the paths converge, this requires coordination between seemingly independent paths.

Some of these problems can be ameliorated. Research has shown how to reduce forwarding state [3], [5], [7], maintain session affinity [5], [7], identify packets whose headers have been changed by a middlebox [3], [6], [7], install forwarding rules for modified packets [7], and allow classification by middleboxes [6]. Yet all these mechanisms add complexity, reduce rather than eliminate some problems, and leave other problems untouched.

The principal contribution of this paper is a detailed exploration of an opposing viewpoint, that *session protocols* might be a better mechanism for service chaining. By *session protocol* we mean any end-to-end protocol, one that establishes and controls communication between end-hosts. There are two major advantages to this approach, which appear in direct contrast to the disadvantages of routing/forwarding above:

- Many of the requirements for service chaining—session affinity, handling modified five-tuples, selective control by middleboxes, and convergence—apply to specific individual sessions. The need for inter-domain control arises primarily because sessions often cross domain boundaries. A session protocol operates on individual sessions rather than on aggregates of them, and can operate end-to-end as well as separately in each domain.
- In the spirit of the end-to-end argument, all of the key functions of a session protocol are performed by *hosts*—whether end-hosts or middlebox hosts. Compared to the session state that is already in these hosts, service chaining requires little additional state. This provides inherent scalability, relieves the pressure on controller capacity, and eliminates the need for network state to do service chaining.

In response to the difficulties with fine-grained forwarding, emerging industry solutions are already replacing fine-grained forwarding with encapsulation, so that forwarding through the service chain is by destination addresses alone [14]–[16]. This is a step in the right direction, but these solutions are intra-domain and some are proprietary. In contrast, we are interested in service chaining that can work across domains and can be added straightforwardly to existing deployments. Session protocols already provide effective and efficient support for mobility [17]–[23] and multihoming [24], [25], and we complete the exploration of this "design pattern" by focusing on middleboxes.

Given the obvious flexibility of signaling in a session protocol, it might be predicted that use of a session protocol for service chaining would provide entirely new opportunities for optimization and network management. This is indeed the case. We introduce a session protocol that does *dynamic reconfiguration*, which means changing the middleboxes in a service chain mid-session. Dynamic reconfiguration could be useful in many situations (see also [26]):

- After directing a request to a backend server, a load balancer could remove itself from the path of the request. The load balancer is no longer a possible point of failure, and there is no need for custom optimizations, like direct server return for response traffic to bypass the load balancer [27].
- A Web proxy cache, ad-inserting proxy, or intrusion detection system could remove itself after its work for a session is done.
- When suspicious traffic is identified, ongoing sessions could be redirected through a packet scrubber for further analysis.
- When the network is congested, video sessions could be redirected through compression middleboxes [28].
- A middlebox that is overloaded or undergoing maintenance, could be replaced with another of the same type (e.g., see [29], [30]).
- When an end-host moves to a new location, a middlebox could be added temporarily to buffer and redirect traffic from the old location. In addition, the old middleboxes in the service chain could be replaced with new ones closer to the new location.

Note that removing a middlebox removes the host *machine* from the path entirely, rather than having the kernel simply bypass the application. This improves performance and reliability, while conserving middlebox resources for sessions that actually need them.

In this paper we describe the Dysco session protocol for service chaining with dynamic reconfiguration. Dysco is an extension to TCP (already a session protocol by our definition) requiring no alterations to end-host applications, middlebox applications, host TCP stacks, or IP routing. Because service chains need not span the entire TCP session, Dysco can be deployed incrementally and across untrusted domains, with conventional security techniques.

We have focused on TCP because of its dominance. Although the Dysco approach will not work for connectionless protocols such as UDP, Dysco does not interfere with forwarding in any way. Therefore existing forwarding solutions can continue to steer all traffic through essential middleboxes such as firewalls, while co-existing with Dysco for more-demanding TCP service chaining.

In addition to design, implementation, and measurement of a Dysco prototype, this paper makes the following contributions:

Highly distributed control: Service chaining and dynamic reconfiguration of the service chain can be performed completely under the control of middlebox hosts. Autonomous operation is valuable not only because it avoids controller bottlenecks, but also because sometimes only the middlebox itself knows which middlebox should be next in the chain for a session, or when its job within a session has been completed. During dynamic reconfiguration, Dysco manages possible contention between different Dysco agents (representing different middleboxes) attempting to reconfigure overlapping segments of the same session at the same time.

Generalized dynamic reconfiguration: For maximum generality, dynamic reconfiguration of a service chain works even if a middlebox being deleted has modified the TCP session, most notably by acting as a session-terminating proxy. It also works if the middlebox has changed the size of a byte stream (e.g., by transcoding or adding/removing content). There is no inherent need for packet buffering except in the case of server migration, when server state must be frozen before it can be transferred. Such packet buffering, when needed, can be performed exclusively by hosts.

Protocol verification: Although the code for dynamic reconfiguration is compact, it was difficult to design, and covers many subtle cases. It would be wrong to assume it is correct without some clear evidence. We have this evidence because we designed the protocol using the modeling language of the model-checker Spin [31], and used Spin to verify it at every stage of design. By presenting an automated proof of correctness, we show how to increase the power of session protocols without sacrificing our confidence in them.

Transparent support for middleboxes: Our prototype intercepts packets in the network interface, so it works transparently with unmodified applications and a wide range of middleboxes. The prototype also supports Linux namespaces, which makes it suitable for virtualized environments (e.g., Docker [32]) and experimentation with Mininet [33].

Dysco allows service chains to span multiple routing and administrative domains, so packets will likely traverse middle-

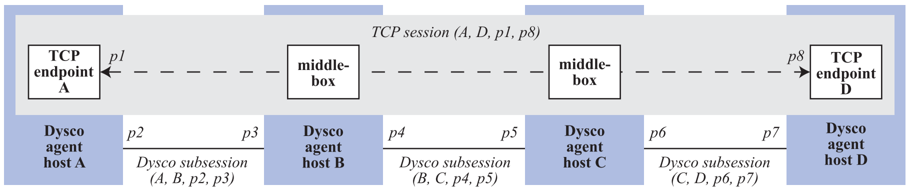

Fig. 1. A TCP session with its Dysco subsessions. The session and each subsession have different five-tuples, including IP addresses and port numbers.

boxes that Dysco does not control. Some of these middleboxes, such as NAT boxes and stateful firewalls, keep track of sequence numbers and drop packets whose sequence numbers fall out of an expected window, breaking the flow of packets and halting the connection. In an improvement over the original version in [34], our reconfiguration protocol makes sure that a TCP subsession looks just like a regular TCP session with consistent sequence numbers, making reconfiguration transparent to the middleboxes in its path.

Scalable implementation: In [34], we presented and evaluated a Dysco implementation using a Linux kernel module that does not scale well, as the kernel imposes a large overhead and cannot process packets at line rate for high-bandwidth links (e.g., 10 Gbps and beyond). For this paper, we implemented the Dysco prototype in BESS [35], a software switch that bypasses the kernel and runs on user space with DPDK [36] for fast packet processing. The new implementation scales better for middleboxes and environments that have to support packet processing at high speeds, e.g., 100 Gbps. Our experiments show that session setup is fast, steady-state throughput is high, and disruption during reconfiguration is small.

II. Dysco Architecture

In Dysco, agents running on the hosts establish, reconfigure, and tear down service chains, relying only on high-level policies and basic IP routing. In this section, we introduce the Dysco architecture and give an overview of the protocol; in §III, we expand on how Dysco can reconfigure an existing service chain.

A. Basic Service Chaining

The basic Dysco concept is that a service chain for a TCP session is a chain of middleboxes and *subsessions*, each connecting an end-host and a middlebox or two middleboxes. There can be any number of middleboxes in the chain. A service chain is set up when the TCP session is set up. The service chain often has the same endpoints as the TCP session, as shown in Fig. 1. Each subsession is identified by a five-tuple, just as the TCP session is. The unmodified end-host applications and middleboxes see packets with the original header of the TCP session; as such, Dysco works with existing application-layer protocols. Fig. 2 shows that the packets go through a TCP stack only at the session endpoints, so that congestion control and retransmission are performed end-to-end as usual. Dysco agents rewrite packet headers for transmission so that packets traveling between hosts have the subsession five-tuple in their headers. In this way, normal forwarding steers packets through the service chain, and there is no encapsulation to increase packet size.

Establishment of the service chain: Establishment of the service chain in Fig. 1 begins when the Dysco agent at host A intercepts the outbound SYN packet addressed to D. If the SYN packet matches a policy predicate, the agent will get an *address list* for the service chain such as $[B, C]$. The agent

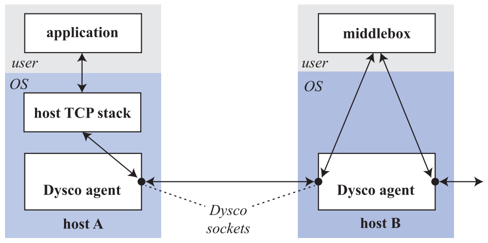

Fig. 2. Data flow inside hosts A and B of Fig. 1.

then allocates local TCP ports for the subsession with the next middlebox. The agent rewrites the packet header with its own address as the source IP address, the address of the next specified middlebox as the destination IP address, and the new allocated TCP ports as source and destination TCP ports. The agent also adds to the payload of the SYN packet the original five-tuple of the session header and the address list $[B, C, D]$. It creates a dictionary entry to map the original session to the new subsession, and another entry to map the subsession to the session on the reverse path. It then transmits the modified SYN packet.

When the Dysco agent at host B receives the SYN packet from the network, it checks to see if the payload carries an address list. If it does, the agent removes the address list from the payload (storing it), and rewrites the packet header with the session information also stored in the payload. The agent also creates dictionary entries to map the subsession to the session and vice-versa, and delivers the packet to the middlebox application. When the SYN packet emerges from the middlebox, the agent retrieves the address list $[B, C, D]$ and removes its own address to get $[C, D]$. It then follows the procedure above to create a new subsession from B to C, rewrite the packet, and transmit the modified SYN packet. This continues along the service chain until the SYN packet reaches D, where it is delivered to the TCP end-host.

When D replies to the SYN, the SYN-ACK packet travels back along the chain of subsessions and middleboxes to continue the handshake. The forward and reverse paths of the TCP session must go through exactly the same middleboxes. Between middleboxes, however, the forward and reverse network paths traversed by subsessions need not be the same.

Middleboxes that modify the five-tuple: If such a middlebox, e.g., a NAT, has a Dysco agent, header modification makes it difficult to associate a SYN packet going into the middlebox with a SYN packet coming out of it. To solve this problem, the Dysco agent applies a local tag to each incoming SYN packet, which it can recognize in the outgoing packet. The agent then associates the incoming and outgoing five-tuples, and removes the tag. (Note that Dysco tags are different from tags in FlowTags [6] and Stratos [7], because they are applied *only* to SYN packets, are *never* sent to the network, and are meaningful only to the agent that uses them.)

A middlebox that modifies the five-tuple, but has no Dysco agent, can also become part of a service chain because ordinary routing of subsession packets directs traffic through it. This will not affect establishment of the Dysco service chain, even though the subsession five-tuple will be different on each side of the middlebox.

Flexible session teardown in each direction: The Dysco protocol preserves TCP's ability to send data in the two directions independently. For instance, one end of a TCP session can send a request, and then send a FIN to indicate that it will send nothing more. It can then receive the response through a long period of one-way transmission. When the TCP session is torn down normally, the chain is torn down along with it. A TCP session can also timeout rather than terminate explicitly, particularly when a middlebox discards its packets, or an end-host fails. In this case the agents will time out the subsessions. If necessary, agents can use heart beat signals to keep good subsessions alive.

B. Role of the Policy Server

We assume that a policy for service chaining combines a pattern that matches five-tuples with an (ordered) list of middleboxes or middlebox types to be traversed by packets matching the pattern. A policy server determines the policies in force, and can optionally trigger dynamic reconfiguration of groups of service chains. Compared to an SDN controller, the policy server has no involvement with individual sessions, and does nothing to enforce its policies (such as installing forwarding rules in network elements).

Selecting the service chain: The first Dysco agent in a service chain needs the policy for the chain. Yet the policy server need not be queried for individual sessions. For example, initial policies can be pre-loaded or cached in Dysco agents. Policies can specify middlebox *types* rather than instances, and agents can choose the instances, e.g., in a round-robin fashion or based on load. In addition, each agent can *add* middleboxes to the untraversed portion of the list. This makes it possible for any agent along the chain to inject policies. This also makes it possible for a middlebox, such as an application classifier, to itself select the next middlebox in the chain. The middlebox communicates its choice to the local Dysco agent, and the agent adds the next middlebox to the head of the policy list.

Initiating reconfiguration of a service chain: In some use cases, Dysco agents initiate reconfiguration of the service chain, without the involvement of the policy server (e.g., when a load balancer or Web proxy triggers the change). In other cases, the policy server is involved, but only in a coarse-grained way. For example, taking a middlebox instance down for maintenance would involve the policy server sending a single command to tell the associated Dysco agent to replace itself *in all of its ongoing sessions*. Similarly, when a measurement system suggests that certain traffic is suspicious, the policy server can send a command to Dysco agents to add a scrubber to the service chain for all sessions matching a particular classifier. The agents handle the full details of reconfiguring the session.

C. Agents Can Reconfigure a Session

The service chain of a session can be reconfigured while the session is ongoing. Reconfiguration operates on a *segment* of the service chain, consisting of some contiguous subsessions and the hosts at their endpoints. When a session is pictured with the client on the left and the server on the right, the Dysco agent at the left end of the segment is called the *left anchor,* and the Dysco agent at the right end of the segment is called the *right anchor.* These terms are illustrated by Fig. 3. Any Dysco agent in the service chain can serve as an anchor, including an agent at a session endpoint.

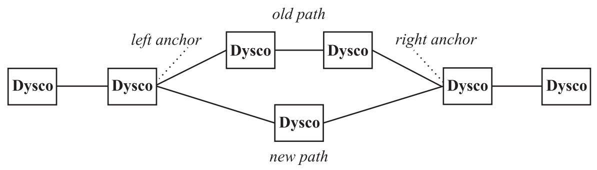

Fig. 3. Agents reconfigure a segment of a session, replacing an old path with two middleboxes by a new path with one.

Fig. 3 shows an old path and a new path, representing the segment before and after reconfiguration. If the old path consists of a single subsession (with no middleboxes), and the new path has at least one middlebox, then middleboxes are being *inserted*. Reverse old and new above, and middleboxes are being *deleted*. If both old and new paths have middleboxes, then the old ones are being *replaced* by the new.

During reconfiguration, the anchors cooperate by exchanging control packets, and they remain in the service chain afterwards. There is no need for packet buffering, because new data can always be sent on one of the two paths.

D. Sessions and Service Chains Need Not Coincide Exactly

In Fig. 1 there is one TCP session and one service chain, and both have the same endpoints. Dysco allows other usages, making it both versatile and incrementally deployable.

When a middlebox modifies the five-tuple, as in §II-A, the service chain spans multiple TCP sessions. For example, a service chain that includes a *session-terminating proxy* (e.g., a layer-7 load balancer, Web cache, or ad-inserting proxy) would encompass two TCP sessions. The Dysco agent of the proxy simply presents data to the proxy application with the TCP session identifier that applies at that point in the service chain. Later, the proxy's work may be completed, e.g., when the load balancer establishes a session to a backend server, or the Web cache realizes the requested content is not cacheable. The Dysco agent can then delete the host from the service chain, in response to a trigger by the proxy. After a session-terminating proxy has been deleted, the resulting service chain would correspond to a single TCP session.

Another option is that a TCP session can be longer than a service chain, or even encompass multiple separate service chains. This is particularly important for *partial deployment* of Dysco or when multiple administrative domains do not trust each other. For example, an end-host that does not run Dysco may connect to the Internet via an ISP edge router that does. This edge router can initiate a Dysco service chain to the remote end-host, or to the other edge of the ISP, on the client's behalf. In another example, a TCP session may access a server in a cloud. The part of the session covered by a service chain in the cloud would begin at some gateway or other utility guaranteed to be in the path of all of the session's packets as they enter the cloud. A Dysco agent in this network element would begin the service chain.

E. Security

Dysco agents use HMACs (Hash-based Message Authentication Codes) [37] with a shared secret key to authenticate and check the integrity of signaling messages exchanged between them. We use the same approach as IPv6 segment routing [38], which also uses HMACs to authenticate and check the integrity of packets between segment routing nodes and assumes the agents share a secret key. Unlike IPv6 segment routing, Dysco authenticates only the signaling messages, as data packets are processed by the Dysco agents only if they belong to an established session. The sending agent uses the SHA-256 cryptographic hash function to create a 32-byte signature of the signaling message. The signature is computed over the content of the payload of the signaling message plus the secret key. Message authentication is optional and can be turned off for faster reconfiguration and session setup.

III. Dynamic Reconfiguration

This section covers the most innovative part of Dysco, which is its protocol for dynamic reconfiguration of an ongoing service chain. First, we give an overview of the protocol's operation. Subsequent sections provide protocol details, organized by significant issues and challenges. Finally, we explain the need for formal verification and how it was accomplished.

A. Protocol Overview

As mentioned in §II-B, reconfiguration can be triggered by policy servers. It can also be triggered by middleboxes in the service chain, e.g., by a load balancer that wants to delete itself after choosing a server and observing that the service chain to the server has been established. Either way, the triggering element must communicate with the left anchor of the reconfigured segment, which initiates the protocol.

Just as the Dysco agent for A in Fig. 1 needs the address list $[B, C, D]$ to set up the original service chain, the left anchor of a reconfiguration needs an address list $[M1, M2, \ldots, rightAnchor]$ with the middleboxes and right anchor of the new path that will replace the old path. The address list may be cached in the anchor agent or provided by the triggering element. When a middlebox triggers its own deletion, it sends a triggering packet to the agent on its left, which becomes the left anchor. The triggering packet contains an address list consisting only of the address of the agent to the middlebox's right, which becomes the right anchor. In this case, after reconfiguration, the new path will consist of a single subsession between the left and right anchors.

Once the left anchor has the address list, it executes the reconfiguration protocol by exchanging control packets with the right anchor. Each control packet carries in its body the associated session identifier. Fig. 4 shows the control packets exchanged by the anchors during the first phase of a simple, successful reconfiguration. The red packets (packets of the first handshake) travel on the old path, so they are forwarded through the Dysco agents of current middleboxes (the *rightDelta* field will be explained in §III-D). The blue three-way SYN handshake sets up the new path within the service chain. As in §II, the SYN carries the address list so that the Dysco agents can include all the addressed middleboxes before the right anchor. During this phase normal data transmission continues on the old path.

In the second phase of reconfiguration, both paths exist. The anchors send new data only on the new path, but continue to send acknowledgments and retransmissions on the old path for data that was sent on the old path. This prevents trouble with middleboxes that might reject packets with acknowledgments for data they did not send. This phase continues until all the data sent on the old path has been acknowledged, after which the old path is no longer used.

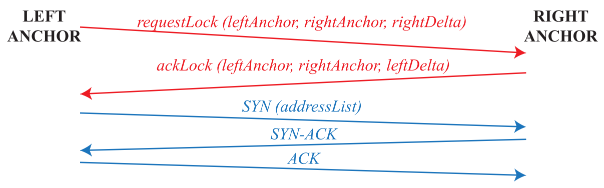
LEFT ANCHOR
RIGHT ANCHOR
requestLock (leftAnchor, rightAnchor, rightDelta)
ackLock (leftAnchor, rightAnchor, leftDelta)
SYN (addressList)
SYN-ACK
ACK

Fig. 4. Control packets exchanged for reconfiguration. Red packets travel on the old path, blue on the new path.

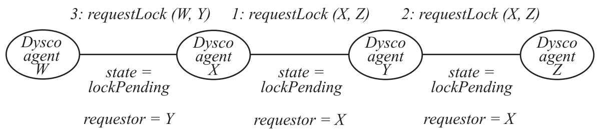

Fig. 5. Contention to reconfigure overlapping segments. The state variables below the subsessions are maintained in the agents at the left ends of the subsessions.

B. Contention Over Segments

Dysco is designed to work even if middleboxes have a great deal of autonomy, so that new solutions to network-management problems can be explored. In the most general case, two different Dysco agents might be triggered to reconfigure overlapping segments at the same time. To prevent this, reconfiguration begins with a request handshake, which is the red (upper) handshake shown in Fig. 4. The handshake is used to prevent simultaneous reconfiguration of overlapping segments, as illustrated by Fig. 5.

For each subsession, the agent on its left maintains a state that is one of *unlocked*, *lockPending*, or *locked*. If its value is *lockPending* or *locked*, then variable *requestor* holds the left anchor of the request for which it is pending or locked.

A left anchor can only send *requestLock(leftAnchor,rightAnchor)* when the subsession to its right is in state *unlocked*. If an agent receives *requestLock(leftAnchor,rightAnchor)* from the left, and the agent is not *rightAnchor*, and its subsession to the right is *unlocked*, then it forwards the packet to the right, while setting the subsession state to *lockPending* and the *requestor* variable to *leftAnchor*. If an agent receives *requestLock* in the same situation except that its subsession to the right is *locked*, it responds to the request with *nackLock*, which propagates back to the requestor—which must abort or delay its reconfiguration attempt. If an agent receives *requestLock* in the same situation except that its subsession to the right is *lockPending*, then it simply waits for further information.

Eventually (see below), a *requestLock* will reach its right anchor, which will reply with *ackLock*. This packet propagates leftward through the segment. As it reaches each agent on its way, the agent changes the subsession state from *lockPending* to *locked*. If the agent is holding a *requestLock* to which it has not yet replied, it now replies to that request with *nackLock*. When the left anchor receives *ackLock*, it begins the second handshake in Fig. 4.

In Fig. 5, a request to lock the segment from X to Z has propagated from X to Z (packets 1 and 2). Meanwhile agent W has been triggered to lock the segment from W to Y. Its request (packet 3) is blocked at X because the subsession to its right is *lockPending*. Eventually X will receive *ackLock*, and reply with *nackLock* to the request from W. If the scenario were different and X received *nackLock* to its own request, it would still reply with *nackLock* to W, so that both X and W would have to abort or try again later. If either tries again later and its intended right anchor has been removed by the previous reconfiguration, its request will be nacked by the agent at the end of the service chain.

This protocol cannot deadlock because of the linear order of the service chain. The rightmost request will never be blocked by a *lockPending* subsession. Therefore it will always receive a reply, which will unblock the blocked request to its immediate left (if any). The unblocked request is now the rightmost request, which will not be blocked again, and so on. Requests could in theory be starved by a continual succession of new requests, but this would not happen in an otherwise correct implementation.

C. Control Signaling

Dynamic reconfiguration requires control signaling, e.g., to resolve contention over segments (§III-B) and to cancel reconfiguration if a new path cannot be created (§III-G).

In [34], we used UDP packets to carry control messages. In this new version of the protocol we use for control a separate TCP session, following the old path, instead. This has the advantage of allowing reconfiguration of service chains that cross NAT boxes or stateful firewalls, which often reject UDP packets. It has the further advantage of leaving the byte stream of the primary TCP session unchanged. The control session is very short, usually consisting of only *requestLock* and *ackLock* messages carried in the payloads of TCP SYN and ACK packets, respectively. These messages carry the session of the session being reconfigured, so the Dysco agents can direct them through the middleboxes of the old path.

A principal design goal for dynamic reconfiguration is to disrupt data transfer as little as possible. We set up a new TCP session on the new path with its own initial sequence numbers, which we set as the last sequence numbers that the anchors see on the old path before initiating setup of the new path. This session looks completely new to any middlebox that is inserted on the new path. While its three-way handshake completes, data transfer continues on the old path.

When the new path is fully set up, the anchors begin to use it for transfer of new data. Meanwhile, the anchors continue to use the old path for acknowlegments and retransmissions of data sent on the old path. After all the data sent on the old path has been acknowledged, the path is no longer used. We do not use an actual FIN handshake to tear down the old path because it is too difficult for the anchors to distinguish between tearing down the old path and tearing down the entire session. This is due to the many possible race conditions between these two cases, which is something revealed by verification (see §III-H). Individual subsessions of the old path can be allowed to time out, and the state kept for them can be discarded.

D. Sequence-Number Deltas

Some middleboxes increase or decrease the size of a byte stream (by transcoding, inserting, or deleting content). They keep track of the difference (*delta*) between incoming and outgoing sequence numbers (a signed integer) in the relevant direction, so that they can adjust the sequence numbers of acknowledgments accordingly. A session-terminating proxy also has a delta because it begins sending in its TCP session to the server with a different sequence number than the client chose. If a middlebox with a delta is deleted, the discrepancy in sequence numbers must be fixed elsewhere.

We make the assumption that once a middlebox is ready for deletion from a session, its deltas do not change.[1] The middlebox's Dysco agent must know the deltas, either through an API or by reconstructing them. As the *requestLock* packet traverses the old path, it accumulates the sum of the middlebox deltas for that direction in the field *rightDelta*. As the *ackLock* packet traverses the old path, it accumulates the sum of the middlebox deltas for that direction in the field *leftDelta*. Each anchor must remember the delta it has received in the *requestLock* handshake.

For the remainder of the session after reconfiguration, each former anchor must apply its delta to packets. Fig. 6 shows how (for the moment ignore all mentions of beta). To simplify the presentation, we assume that sequence numbers do not wrap around to zero.

In Fig. 6, whenever a packet is coming into the middlebox co-located with a former anchor, from the new path (reconfigured segment), the Dysco agent adds its delta to the sequence number (SN in the figure). This simulates what the old path would have done if its middleboxes were still present. To balance this adjustment, at the same point in the session path, but in the opposite direction, the same Dysco agent subtracts its delta from the acknowledgment number (AN in the figure). The invariant preserved by these transformations is that all middleboxes present in both old and new paths (not to mention the endpoints) see continuous sequence and acknowledgment numbers before and after reconfiguration.

E. Sequence-Number Betas

Using the old path for data transfer while the new one is being set up is an important feature of our protocol. It means that reconfiguration does not delay data transfer, and does not require buffer space anywhere (except in cases where the state of a middlebox in the old path must migrate to a middlebox in the new path, making buffering inevitable). However, this desirable feature creates the need for another adjustment of sequence numbers.

The problem is that an anchor must choose an initial sequence number for the new path before it has finished sending on the old path. When the new path is ready for use, the next sequence number sent may be greater than the next sequence number expected on the new path by a number *beta,* because beta bytes were sent on the old path while the new one was being set up. This will not affect the sequence and acknowledgment numbers observed by middleboxes that were previously present, but new middleboxes may see a large gap. Some middleboxes keep track of sequence numbers (e.g., Linux iptables firewalls), and would halt data transfer on the new path because of this apparent error.

To remove the sequence-number gap, each anchor must inform the other how many bytes have been sent on the old path after they have selected their initial sequence numbers for the session setup. Informing the other anchor is tricky. If an anchor sends its beta in a control message, it can be lost, or data packets can arrive on the new path before it arrives on the old path. Also, we do not want to include a new option in the data packets to avoid increasing packet size and consequently having to fragment them.

To transmit betas, we use a property of the timestamp option that says that its value must be monotonically non-decreasing.

[1]Without this assumption, there must be a wait while the last data passes through the old path, during which new data cannot be sent on either path.

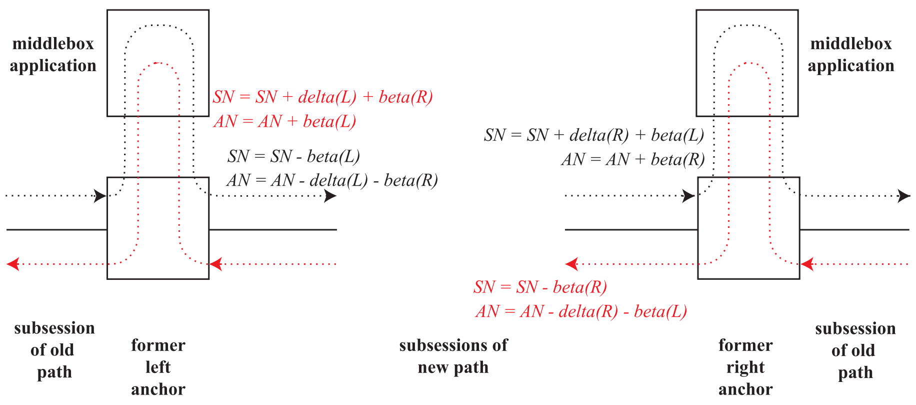

Fig. 6. How former anchors modify TCP packets (dotted arrows) of new subsessions. Black refers to the forward path, while red refers to the reverse path.

When we send the TCP SYN or SYN+ACK between the anchors, we set the value of the timestamp option (SEG.TSval) to the last SEG.TSval sent on the old path. After we switch to the new path, the anchors add their betas to the initial timestamp values in all data packets until they receive an ACK on the new path. The ACK confirms that the other anchor has received the piggybacked beta. Note that the timestamp values will be the same from the moment we switch to the new path until the anchors receive an ACK on the new path. Finally, the anchors have to translate the timestamp options before the messages leave the new path to prevent the end hosts from seeing a non-decreasing sequence of timestamp values. Even though the timestamp option is not mandatory, all major operating systems enable it by default to cope with today's high-speed networks. In the unlikely case that the timestamp option was not negotiated on the old path, the Dysco agents can negotiate it on the new path. But, in this case, fragmentation and reassembly on the new path are unavoidable, as the timestamp option must also be included in the data packets.

The use of betas from the left and right anchors (*beta(L)* and *beta(R)*, respectively) is also shown in Fig. 6. As a packet enters the new path from an anchor, the anchor subtracts its own beta from the sequence number. As the packet exits the new path going into a middlebox co-located with the other anchor, the anchor adds the other anchor's beta. Clearly this affects sequence numbers observed within the new path only. To balance these adjustments, at the exact same points in the path, wherever the Dysco agent adds [subtracts] a beta to a sequence number, it subtracts [adds] the same beta from the acknowledgment number in the other direction.

Do the deltas and betas interact with each other? Fortunately not, because they affect the data stream in different places. In Fig. 6, consider adjustments to the sequence numbers in the forward direction. The left anchor subtracts *beta(L)* so it will be observed by middleboxes within the new path. The right anchor adds *beta(L)* to cancel what the left anchor did, and adds *delta(R)* to the sequence numbers to be observed outside the new path. Similarly, concerning sequence numbers in the reverse direction, the right anchor subtracts *beta(R)* so it will be observed by middleboxes within the new path. The left anchor adds *beta(R)* to cancel what the right anchor did, and adds *delta(L)* to the sequence numbers to be observed outside the new path.

F. Packet Handling on Two Paths

In the second phase of reconfiguration, both old and new paths exist. To handle packets correctly, the anchors must decide which path to use when sending data or acknowledgments, and must know when the old path is no longer needed. The following algorithm is independent of deltas and betas. It should be visualized as executing "in the middle" of the anchors as they perform reconfiguration. The delta and beta adjustments should be visualized as executing "at the interface" between the anchors and the new path. So, for example, the black adjustments in Figure 6 are applied to packets by the left anchor after middlebox processing and after the algorithm in this section. The black adjustments are applied by the right anchor before the algorithm in this section and before middlebox processing.

To make these decisions, an anchor maintains the following variables (the "plus one" follows TCP conventions for sequence numbers):

- *oldSent:* highest sequence number of bytes sent on old path, plus one (this is known at the beginning of the phase, as no new data is sent on the old path);
- *oldRcvd:* highest sequence number of bytes received on old path, plus one;
- *oldSentAcked:* highest sequence number sent and acknowledged on old path, plus one;
- *oldRcvdAcked:* highest sequence number received and acknowledged on old path, plus one;
- *firstNewRcvd:* lowest sequence number received on the new path, if any.

A byte sent by an anchor is allocated to a path according to the following rules. If a packet contains data for both paths (both new and retransmitted bytes), then the data must be divided into two new packets.

predicate on *byteSeq*	where to send byte
$byteSeq < oldSent$	old path
$byteSeq \geq oldSent$	new path

Acknowledgment numbers are a little different because their meaning is cumulative. For these the rules are:

predicate on *packetAck*	where to send ack
$packetAck \leq oldRcvd \wedge packetAck > oldRcvdAcked$	old path
$packetAck > oldRcvd \wedge oldRcvd = oldRcvdAcked$	new path
$packetAck > oldRcvd \wedge oldRcvd > oldRcvdAcked$	new path, also ack *oldRcvd* on old path

If the two sets of rules imply that the data of a packet goes to one path and its acknowledgment goes to another, then the packet must be divided into two. These rules need not consider deltas, as deltas are already applied to incoming packets, and not yet applied to outgoing packets.

For an anchor to decide that it no longer needs the old path, of course it must have received acknowledgments for everything it sent on the old path, or $oldSentAcked = oldSent$. Knowing that it has received everything on the old path is harder, unless it has received a FIN on the old path, because it does not have direct knowledge of the cutoff sequence number at the other anchor. The first byte received on the new path is not a reliable indication, because earlier data sent to it on the new path may have been lost. The correct predicate is:

$$oldRcvdAcked = oldRcvd \wedge oldRcvd = firstNewRcvd$$

The first equality says that everything received has been acknowledged. The second says that the cutoff sequence number must be *oldRcvd*. When the old path is no longer needed, reconfiguration is complete.

If a stateful middlebox in the session is being replaced, additional delay must be introduced. First, all use of the old path must be completed. Second, the stateful middlebox on the old path must export its state for that session to the new stateful middlebox, using existing mechanisms [30]. Then and only then can data be sent on the new path. During the interval when the old path is being emptied and state is being migrated, the anchors must buffer incoming data. Note that the Dysco protocol does not use buffering during a reconfiguration, but in this case it needs to buffer the packets to prevent them from arriving at a middlebox before its state has been migrated. We evaluated middlebox replacement with state transfer in [34].

G. Failures

The old path must be fully operational for reconfiguration to work. If the old path fails during reconfiguration, then the entire session will be lost, which is exactly what would happen if there were no Dysco and no reconfiguration. Unfortunately, this means that dynamic reconfiguration cannot be used to recover from the failure of a middlebox. The utility of reconfiguration is limited to policy change and resource management, rather than fault-tolerance.

The most significant failure during reconfiguration is failure to set up the new path, which can happen because of host failure or network partition. The remedy is to cancel the reconfiguration, so the session continues to use the old path. A cancellation handshake through the control session on the old path (§III-C), initiated by the left anchor, unlocks the links of the old path and informs the right anchor.

If control packets are lost, then the protocol detects this and retransmits them. The control session is used only for a short time, to initiate reconfiguration and possibly to cancel it if the new path cannot be set up. After this, the session is allowed to time out.

H. Design and Verification

In designing the original reconfiguration protocol, we had to solve a number of related problems simultaneously. We had to decide how to make the cutoff between the old and new paths for maximum efficiency (§III-A), how to exercise distributed control among conflicting reconfiguration attempts (§III-B), how to compute and use deltas to accommodate the broadest range of middlebox applications (§III-D), how to split acknowledgments across the two paths and determine when the use of the old path is completed (§III-F), and how to handle failures (§III-G). We had to decide whether any particular packet should be TCP or UDP (§III-C) [34]. We also had to deal with many race conditions—for example, an anchor might receive a FIN going in either direction in almost any state, and the FIN might indicate the completion of data transmission on the old path or the completion of end-to-end TCP data transmission.

We did not believe that we could design such a protocol correctly without help, so we designed it in Promela, which is the modeling language of the model-checker (verifier) Spin [31]. In Promela, each Dysco agent is a concurrent process that communicates with other processes through message queues. The messages represent both TCP and UDP packets, with fields for sequence numbers and other metadata. Each agent is structured as a finite-state machine that can react to the receipt of a message by reading and writing local variables, sending other messages, and/or changing state. Choices made by end-hosts and middlebox applications are modeled by nondeterminism in the program. As a result, the Promela program for a Dysco agent has a straightforward structure that translates easily to actual implementation code.

The great advantage of using Promela for design is that we were able to verify the model at every step, obtaining immediate feedback on bugs and unresolved issues. It was necessary to verify each configuration separately, where a configuration is an initial service chain and a set of attempted reconfigurations. For each configuration, Spin checks the model for all possible executions, meaning all possible network delays and scheduling decisions, which in turn generates all possible interleavings of modeled events. In a typical verification run for a typical configuration, Spin constructs a global state machine of all possible execution behaviors with 100 million state transitions.

What can verification tell us? Any run of Spin will find errors such as deadlocks and undefined cases. In addition, it is possible to check stronger properties by putting assertions at appropriate points in the model. If execution reaches an assertion point and the assertion evaluates to false, that will also be flagged as an error. Using this technique, we were also able to verify that each configuration has the following desirable properties:

- When multiple left anchors contend to lock overlapping segments, exactly one of them succeeds.
- No data is lost due to reconfiguration.
- Unless the new path cannot be set up, an attempted reconfiguration always succeeds.
- The sequence and acknowledgment numbers received by end-hosts are correct.
- The original TCP session terminates cleanly.

In the new version of the protocol described in this paper, the only significant change is the addition of beta adjustments as described in §III-E. Fig. 6 and the explanations based on it in §III-D, §III-E, and §III-F form a "proof by picture" that the beta adjustments are correct and independent of other aspects of the modeled protocol.

The model, along with extensive documentation of design, modeling abstractions, and Spin runs, can be found at [39]. It shows that with modern tools, protocol design can be more ambitious without sacrificing robust operation.

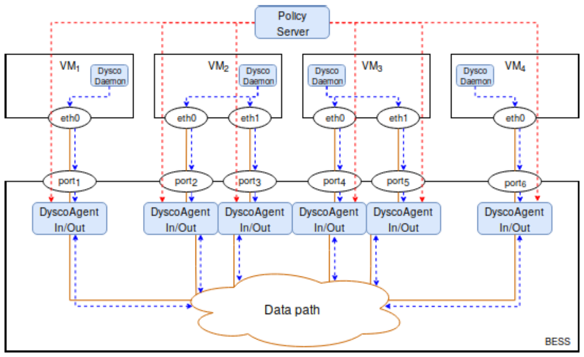

Fig. 7. Dysco prototype implementation, where solid orange lines represent the data path, blue dashed lines the control path, and red dashed lines the management path for distributing policies.

IV. Dysco Prototype

Our Dysco prototype consists of user-level BESS modules[2] that work on the data path (agent) and control path (daemon). The Dysco daemon communicates with an external policy server to receive the policies the agents must enforce. Fig. 7 shows the high-level architecture of our implementation.

A. Dysco Components and Interfaces

Agent: The Dysco agent supports unmodified end-host applications, middleboxes, and host network stacks by intercepting packets going to/coming from the network. The agent could be implemented in various ways, including a modified device driver, a Linux kernel module [34], or a software switch. In [34], we implemented Dysco in a Linux kernel module that received and sent packets directly to the NIC driver. However, the Linux kernel is not fast enough to cope with the current high-speed networks, as it can barely handle a million packets per second. In this new implementation of the protocol, the Dysco agents are user-space BESS modules that intercept packets at the ports exported by the BESS switch. We decided to use BESS because of its modularity and to provide fast packet processing using DPDK. Also, by intercepting packets at the port level, we can support applications and middleboxes transparently and do not have to change the NIC drivers. As the Dysco agent processes all packets from a TCP session, it can change how TCP behaves in several ways. For example, it can advertise a smaller receive window to throttle a sender during reconfiguration or even prevent it from sending data at all by advertising a window of size zero. Our prototype also supports network namespaces for virtualized environments, such as Docker and Mininet.

Middleboxes: Dysco supports unmodified middlebox applications, and we have successfully run with NGINX [41], Iptables/Netfilter [42], Linux tc [43], and libpcap-based middleboxes. Most middleboxes send and receive data via `libpcap`, user socket, Linux `sk_buff`, or DPDK. Some middleboxes only read the packets (e.g., PRADS [44], Bro [45], Snort [46], Suricata [47], Linux tc [43], Iptables/Netfilter Firewall [42]) while some others modify the TCP session identifier or sequence numbers (e.g., Iptables/Netfilter NAT [42], HAProxy [48], Squid [49]). Middleboxes that only read the packets and use `libpcap`, `sk_buff` or DPDK run transparently and unmodified with Dysco. To support the removal of TCP-terminating applications (e.g., load balancers), we provide a library function (`dysco_splice`) that a (modified) middlebox can use to trigger its removal. Dysco also supports middleboxes that can import and export internal state as part of migrating a session from one middlebox instance to another, inspired by OpenNF [29].

Daemon: The Dysco agent performs session setup and teardown, as well as data transfers. As we now use only TCP packets in our protocol, locking and reconfiguration operations are triggered by a user-space daemon but are processed by the agents directly on the data path. To speed up the identification of the Dysco control packets on the data path, we use TCP segments with the URG and ACK flags set to one and with option 254 (reserved for experimentation). The control packets carry information to identify the left and right anchors for the locking phase, and a service chain for setting up the new path.

Policy server: The policy server provides a simple command-line interface for specifying the service-chaining policies and trigger reconfiguration of live sessions. A policy includes a predicate on packets, expressed as BPF filters, and a sequence of middleboxes. The policy server distributes these commands to the relevant Dysco agents. Commands can be batched and distributed to different hosts using shell scripts. The policy server and the Dysco agent and daemon consist of over 7,500 lines of C and C++. The source code of Dysco as well as the shell scripts used for the evaluation are available at [39].

B. Protocol Details

Tagging SYN packets: The local tags added to SYN packets, as described in §II-A, are implemented with TCP option 253 (reserved for experimentation). The option carries a unique 32-bit number to identify the session. SYN packets are tagged only when they are inside a middlebox host.

Packet rewriting for data transmission: During data transmission, the agent simply rewrites the five-tuple of each incoming or outgoing TCP packet and applies any necessary sequence and acknowledgment number delta, timestamp delta, and window scaling. Since the agent rewrites the packet header, it has to recompute the IP and TCP checksums. All checksum computations are incremental to avoid recomputing the checksum of the whole packet.

TCP packets for reconfiguration: The agents implement the reconfiguration protocol using TCP packets. We decided to use TCP in this new implementation to allow NAT crossing, a limitation of our previous implementation [34]. We use TCP SYN and SYN+ACK segments to carry the control messages. The control messages carry the five-tuples of the TCP sessions going through the reconfiguration so that the Dysco agents can associate the control message with the session state. If a middlebox is inserted during the reconfiguration, the Dysco agent removes the payload, tags the SYN packet, rewrites the session, and forwards the SYN segment to the host. We forward the three-way handshake segments to the hosts so the middleboxes in the new path can create the necessary state for the session as if it were a new session in that segment of the service chain.

Triggering a reconfiguration using "splice": To deal with middleboxes that terminate TCP sessions and want to remove themselves, Dysco offers a library function that receives two sockets and a delta representing how much data was added to or removed from the first socket before delivering the data to the second socket:

```
intdysco_splice(intfd_in, intfd_out, intdelta)
```

[2]These modules can be easily ported to Click modules [40], for instance.

A positive delta indicates that data were added to, and a negative delta indicates that data were removed from `fd_in`. This option requires the modification of a middlebox to call the library function. In [34], we also discussed a module for transparently removing middleboxes that use the Linux's "splice" system call.

Differences in TCP options for two spliced TCP sessions: When a Dysco agent initiates a "splice" of two TCP sessions, the Dysco agents on the left and right anchors need to translate not only the sequence and acknowledgment numbers of each packet but also the TCP options that differ between the two sessions or have a different meaning. The relevant options are window scaling, selective acknowledgment, and timestamp. Window scaling is easy to convert, as the anchors record the scale factor negotiated during the session setup. The Dysco agent first computes the actual receiver window of a packet using the scale factor of its incoming subsession and then rescales the calculated value by the scale factor of the outgoing subsession. The translation of the selective acknowledgment (SACK) blocks is particularly important because the blocks of one session have no meaning to the other session (if blocks are not translated, the Linux kernel will discard all packets that contain blocks with invalid sequence numbers). To convert the sequence numbers of SACK blocks, the anchors add to (or subtract from) each sequence number the delta that they receive during session reconfiguration. Timestamps are used for protection against wrapped sequence numbers and RTT computation. The Linux kernel keeps track of the highest timestamp received and discards packets whose timestamps are too far from it. To avoid packets being discarded by the kernel, Dysco translates timestamps in the same way as it does with sequence numbers.

NAT crossing: Because we used UDP packets to send control messages, a critical limitation in our previous implementation [34] was that we could not reconfigure a session going through a NAT box. The use of TCP packets helps us solve this problem, but we still need to identify segments of a session that cross a NAT box. Now, each agent inserts the five-tuple of the subsession in the payload of the TCP SYN segment of a reconfiguration message. When the agent on the other end of the subsession receives the control message, it compares the subsession in the payload with the five-tuple of the packet. If they differ, the sending agent is on the private side of a NAT. The agent on the public side of the NAT has to change the private IP address and TCP source port of the session to their correspondent public values that are in the packet header before forwarding the packet to the application and the subsequent hosts along the service chain. On the returning SYN+ACK segment, the public session is propagated back to the hosts that are on the private side of the NAT. We use the public session as the session identifier in all locking and reconfiguration messages. If a reconfiguration crosses a NAT box, only a host on the private side can start the reconfiguration.

V. Performance Evaluation

We evaluate Dysco in the three main phases of a session across different network settings. First, we measure the latencies for session setup to quantify Dysco overhead, which includes processing middlebox address lists in the SYN packets and signing and checking the control messages with HMAC. Second, we measure the throughput of a session during normal data transfer to show that the Dysco agents can forward packets at high speed. Third, we show that dynamic reconfiguration improves end-to-end performance and introduces minimal transient disruptions. Finally, we evaluate reconfigurations that cross NATs and stateful firewalls.

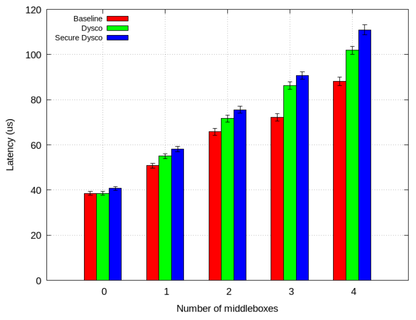

Fig. 8. Latencies for session initiation.

The testbed consists of three servers with two Intel Xeon Silver 4114 (10-cores @2.2GHz), each with 96GB of RAM and two Mellanox ConnectX-5 EN 100Gbe NICs connected to an Edge-Core Wedge 100BF-32X switch that performs either layer-two forwarding or layer-three routing, depending on the experiment. We use the BESS software switch [35] and DPDK 18.08 for composing the modules that implement the data path between the hosts. Finally, we use `qemu-v2.11.1` for hardware virtualization.

A. Session Initiation

Fig. 8 shows the session setup latency between client and server under three scenarios: without Dysco, with Dysco, and secure Dysco using HMAC for signing and checking the control messages. The number of middleboxes varies from 0 to 4. The measurements represent the time for a TCP socket `connect()` at the client, which is the round-trip for establishing the TCP session to the server with a confidence interval of 95%. The client and server are virtual machines that run on different servers, and all four middleboxes are virtual machines that run on the third server. The middleboxes simply route the IP packets using the Linux kernel.

On regular Dysco, the agents perform lookups, rewrite packet header fields, and recalculate checksums for each SYN and SYN-ACK segment along the service chain. In the secure Dysco, the agents also have to check the integrity of the received messages with the HMAC code and sign the outgoing messages after the changes in each hop along the service chain. The overhead introduced in the worst case (secure Dysco $\times$ no Dysco) is only $22\mu s$ when we have four middleboxes. Note that the RTT in the testbed is under $110\mu s$ in the worst case, so the overhead is insignificant in real scenarios where RTTs are expected to be in the order of milliseconds.

B. Data-Plane Throughput

We first evaluate the ability of the Dysco agents to saturate a 100 Gbps link, using one server as the sender and another as the receiver. We create 60k different TCP sessions between the sender and the receiver. Figs. 9 and 10 show the number of packets per second (left y-axis) and the throughput (right y-axis) as a function of the packet size. We vary the packet sizes from 64 bytes up to 1518 bytes and use one logical core (Fig. 9) and 16 logical cores (Fig. 10). Because we assume servers in the wild use logical cores as well as physical cores, we do not disable hyperthreading for this experiment as it is the common practice in performance evaluations

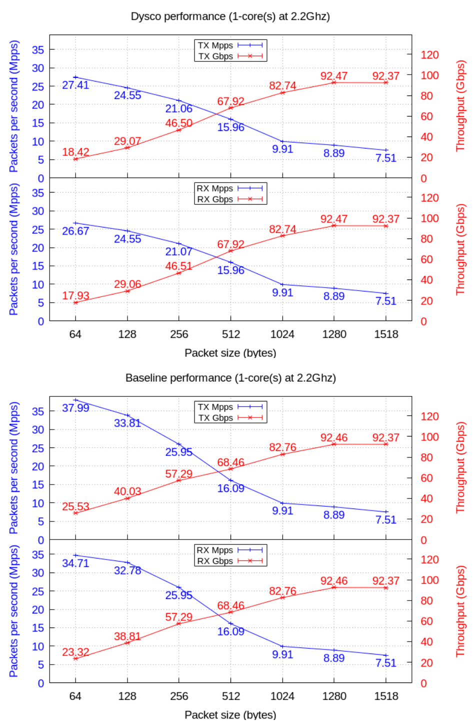

Fig. 9. Performance of Dysco using one logical core.

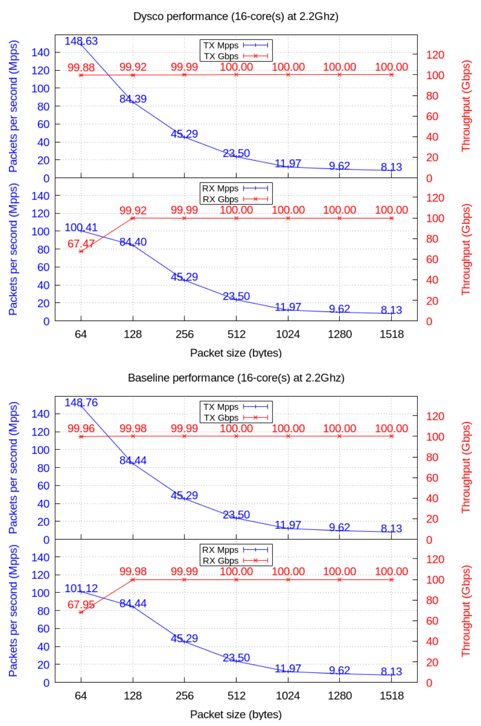

Fig. 10. Performance of Dysco using 16 logical cores.

that measure packets per second. Dysco introduces a performance penalty of up to 27.85% with one logical core and 64-byte packets, because table lookup, packet modifications, and checksum computation are expensive tasks that the Dysco agents have to perform in each packet. Wang *et al.* show that hash table lookup is the most time-consuming stage during packet processing [50]. We use the Google dense hash map for the lookup operation. Performance improvements in this data structure will automatically translate to performance improvements in Dysco. However, Fig. 10 shows that Dysco saturates a 100Gbps link with 128-byte packets and 16 cores.

We also measured the number of requests that NGINX [41], a popular HTTP server, can sustain under Dysco, and compared the results with the baseline. The measurement was performed with `wrk` [51], an HTTP benchmarking tool, with 16 threads and 400 persistent connections, as recommended in [51]. NGINX can serve more than 180,000 connections per second when the client and the server are connected directly, and a little under 175,000 connections per second when four middleboxes are between the client and the server. The largest difference between Dysco and the baseline is less than 4.08%.

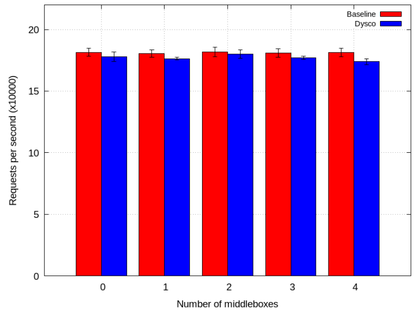

Fig. 11. Number of HTTP requests per second NGINX can serve under Dysco and the baseline.

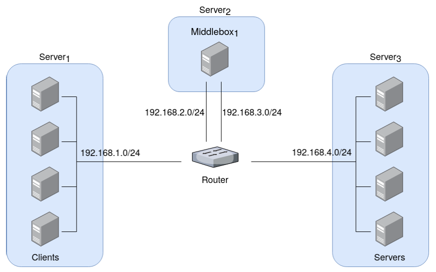

Fig. 12. Testbed topology for the performance evaluation of the reconfiguration experiments.

C. Dynamic Reconfiguration

In this section, we investigate a few scenarios of dynamic reconfiguration. We use the logical topology of Fig. 12; the clients, servers, and TCP proxy are virtual machines running on different physical servers.

We run TCP sessions from four Clients to four Servers, passing through the Router and Middlebox$_1$, which is running a TCP proxy. After 40, 60, 80, and 100 seconds, we trigger reconfigurations that remove Middlebox$_1$ from a client-server pair and direct the traffic of all TCP sessions between them directly from the client to the server passing only through the Router. Each client-server pair has a bundle of 125 TCP sessions for a total of 500 simultaneous sessions.

The top of Fig. 13 shows the goodput before and after each reconfiguration. The time series represents measures of application data (goodput) at one-second intervals. After each reconfiguration, the goodput of the sessions that no longer go through the proxy increases significantly. We can see that after 100 seconds when all 500 sessions no longer go through the proxy, the overall goodput increases from the time interval before the reconfigurations started. The bottom of Fig. 13 shows the CPU utilization at the proxy. We can see that the CPU utilization decreases at the instants 40, 60, 80, and 100, going to zero after all the reconfigurations end.

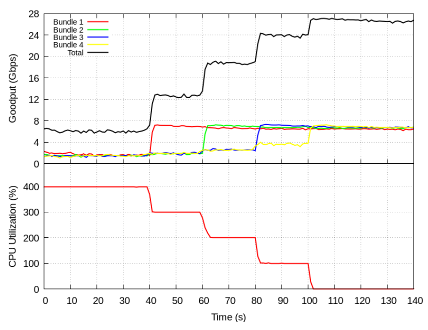

Fig. 13. Goodput of TCP sessions (top) and CPU utilization of the proxy (proxy) before and after multiple reconfigurations.

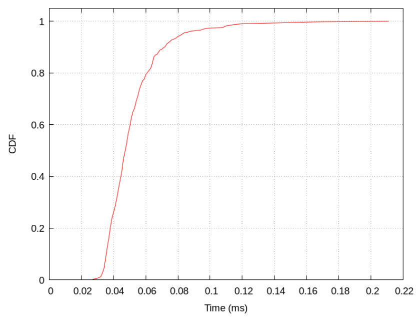

Fig. 14. CDF of the reconfiguration time for the proxy removal.

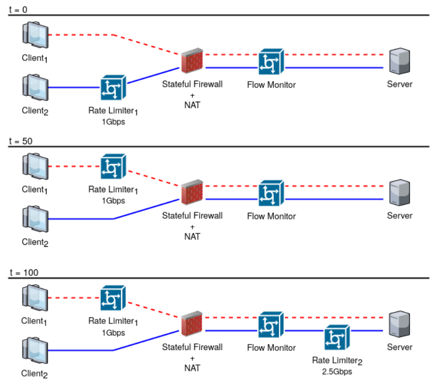

Fig. 15. Stateful firewall evaluation.

Fig. 13 shows that the reconfigurations are successful and the traffic reaches steady-state behavior after each reconfiguration. The Dysco agent on the proxy advertises a small window to the senders during reconfiguration to reduce the amount of traffic on the receivers. Note that during reconfiguration, packets are received from both paths causing a surge of traffic at the receivers. We initially tested a zero window advertisement, but the performance degraded significantly. The best strategy was to advertise the minimum of the actual advertised window and a small constant (16K), allowing the flow of packets to continue without overwhelming the receivers. Fig. 14 shows that reconfiguration time is short: almost 80% of reconfigurations took less than 0.06ms, and the worst time was less than 0.22ms. The larger values happen when control messages are lost and need to be retransmitted.

Finally, we evaluate Dysco when a reconfiguration has to go through a NAT box and a stateful firewall that do not run Dysco agents. Fig. 15 shows the topologies we use in the evaluation in different moments before and after reconfigurations. The client, server, and middleboxes are virtual machines

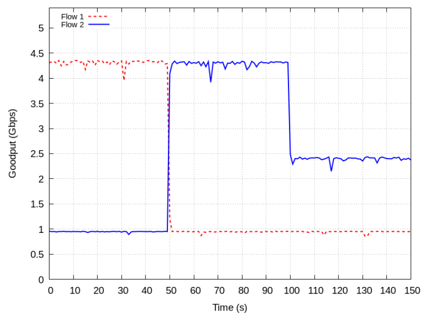

Fig. 16. Inserting and removing middleboxes through a NAT.

running on different physical servers. They use Dysco agents to create the service chain and to reconfigure an ongoing session. Each client establishes a TCP session with the same server, and both sessions cross a stateful firewall, a NAT box, and a flow monitor. The clients send data to the server using *iperf*. The $RateLimiter_1$ and $RateLimiter_2$ forward packets and use Linux `tc` [43] to limit the bandwidth to 1 Gbps and 2.5 Gbps, respectively. The firewall uses `iptables` with the `conntrack` module enabled and sequence number tracking.

Fig. 16 shows the goodput of both sessions before and after a few reconfigurations. At time $t = 0$, $Client_1$ sends packets at its maximum bandwidth (red dashed line) while $Client_2$ sends packets through $RateLimiter_1$ (blue line). We trigger a reconfiguration at $t = 50$ in which we remove $RateLimiter_1$ from the $Client_2 \leftrightarrow Server$ session and insert it into the $Client_1 \leftrightarrow Server$. In this case, we can see that the goodput of the first session (red dashed line) decreases to 1 Gbps while the goodput of the second session (blue line) increases to $\sim$4.4 Gbps. At time $t = 100$, we trigger a new reconfiguration in which we insert $RateLimiter_2$ to the $Client_2 \leftrightarrow Server$ session. In this case the goodput decreases to 2.5 Gbps. Fig. 16 shows that the reconfigurations do not disrupt the sessions, and a steady state is reached afterwards.

The experiments in this section highlight the new features of the Dysco protocol as well as the performance improvements of a user-level implementation with DPDK. In [34], we show results for reconfiguration with state migration, and other performance results specific to the kernel-level implementation.

VI. Related Work

A. Service Chaining by Forwarding

The vast majority of service-chaining implementations and research proposals use forwarding to steer packets through middleboxes. The motivation for the CPR tool [52] reminds us just how error-prone this can be! Here we mention a few of these schemes with special relevance to Dysco.

BGP: Early solutions to dynamic service chaining manipulate BGP to "hijack" traffic, either within a single domain [53] or across the wide area [11]. But manipulating BGP is risky in the wide area, and it operates coarsely on destination IP prefixes rather than individual sessions. Plus, it is difficult to use BGP to insert multiple middleboxes in a service chain.

Stratos and E2: Stratos [7] and E2 [54] are designed for middlebox deployment within clouds. They use fine-grained forwarding rules for (static) service chaining, inheriting the scaling challenges mentioned in §I. They also offer integrated solutions for managing middleboxes, including elastic scaling of middlebox instances, fault-tolerance, and placement. Dysco is not concerned with middlebox management and can be readily combined with any approach to it.

OpenNF: OpenNF [29] (and also Split-Merge [30]) assumes that dynamic service chaining is provided by updating how SDN switches forward packets. The special contribution of OpenNF is efficient, coordinated control of forwarding changes and middlebox state migration, so that middleboxes can be replaced quickly and safely. Our Dysco prototype was easily extended to support importing and exporting middlebox state. As a session protocol, Dysco can naturally handle a wider range of reconfiguration scenarios than OpenNF can, including removing proxies. OpenNF is designed for use in an SDN environment, while Dysco places no constraints on the choice of the control plane. Also, there is a risk of performance problems with OpenNF controllers because they are responsible for packet buffering.

B. Service Chaining by Session Protocols

DOA: Like Dysco, DOA [55] uses a session protocol for service chaining. Dysco and DOA differ as follows: (i) DOA requires a new global name space, while Dysco does not; (ii) DOA does not support dynamic reconfiguration of the service chain; (iii) DOA inserts middleboxes only on behalf of end-hosts (ignoring those inserted on behalf of administrators), and (iv) DOA uses encapsulation, so that both high- and low-level addresses are included in each packet. This increases packet size, which may cause MTU problems.

NUTSS: In the NUTSS architecture [56], session setup begins with an end-to-end handshake between end-hosts with high-level names. The handshake signals are routed by the high-level names through an overlay network of servers. These servers are not middleboxes, however, but rather policy servers that provide name authentication, negotiation of encryption, and distribution of credentials. After the handshake in the overlay network, packets in the ordinary network carry credentials they can use to be accepted by middleboxes such as firewalls that they are routed through. NUTSS requires changes to all end-hosts and middleboxes.

Connection Acrobatics: Nicutar *et al.* [57] use Multipath TCP to insert middleboxes into sessions. However, middleboxes cannot be inserted until a TCP session is established end-to-end. Subsequently a second end-to-end path is established going through a middlebox, and the first path is removed. A second middlebox can then be inserted between an end-host and the first middlebox, and so on. This approach takes dynamic insertion too far—because middleboxes are not included as the session is formed, middleboxes cannot protect an end-host from unwanted sessions as a firewall does, cannot choose the end-host of a session as a load balancer does, and are not guaranteed to see all packets within a session.

NSH: Network Service Header [14] is an encapsulation format for service chaining without the use of forwarding rules, so in this list it is most closely related to DOA. NSH is an intra-domain format only, and there is no mechanism for dynamic reconfiguration.

C. Research Complementary to Dysco

Mute [58] is a technology for utilizing resources in multiple edge clouds, and its use is made possible by Dysco.

Encrypted content: Multi-context TLS (mcTLS) [12] enables middleboxes to operate on encrypted traffic, through a signaling protocol that (i) establishes a TCP session for each hop in the service chain and (ii) exchanges keys for decrypting and reencrypting the data. Like Dysco, mcTLS has a list of middleboxes in a session setup message. In mcTLS, however, the list is carried in the TLS Hello message rather than the TCP SYN packet. mcTLS illustrates clearly that if middleboxes are to operate on encrypted sessions then they must receive encryption keys through the session protocol. Fine-grained routing and forwarding can never be sufficient to enable such middleboxes to do their jobs.

Mobility and multihoming: End-to-end signaling protocols have been widely used for supporting end-host mobility [1]. Of these, ECCP [17], TCP Migrate [18], Quic [59], and msocket [60] are TCP-oriented. ECCP, TCP Migrate, and Quic are oblivious to middleboxes. msocket explicitly uses signaling at the application layer to deal with the complexities introduced by middleboxes. Likewise signaling protocols have been used for supporting multihoming, notably ECCP [17] and Multipath TCP [24]. All of these protocols are intrinsically compatible with Dysco, which suggests that merging the approaches would be fruitful.

Middlebox implementation: There has been much recent research on making middleboxes more efficient, particularly those that depend on TCP session states or reconstruct TCP byte streams. *Microboxes* [61] re-arrange the functional modules of a collection of middleboxes, so they are compatible with Dysco when used within what Dysco sees as a single middlebox. *mOS* [62] provides new implementation abstractions, e.g., for monitoring TCP states, so might be helpful for producing middleboxes with integrated Dysco agents.

VII. Conclusion

In this paper we have presented motivations for using a session protocol as the mechanism for TCP service chaining. Our Dysco protocol meets the requirements of a wide variety of use cases. The protocol interoperates smoothly with the use of routing and forwarding for service chaining, so there is no need to exclude either approach.

Dysco introduces a very general capability for dynamic reconfiguration of a service chain, along with a number of use cases for it (§I). Correctness of this capability has been formally verified, including the property that no data is lost due to reconfiguration. Concerning the demand for new capabilities such as dynamic reconfiguration, the question to ask is not, "Is this capability being demanded now?", when even much simpler things are difficult to deploy. A fairer question might be, "Would good uses for this capability be found if it were readily available?"

Because Dysco agents have a great deal of autonomy, the load on centralized policy servers is relatively light. Our experiments show that session setup and teardown are fast, steady-state throughput is high, and disruption due to dynamic reconfiguration is minimized. Many middleboxes can run unmodified in the Dysco architecture. Future work will include more measurements, prototyping of new use cases, and deployment of Dysco in a real network.

Some limitations remain, particularly in the realization of Dysco's potential for inter-domain service chaining. However, the Dysco approach has received far less attention than fine-grained forwarding, which cannot be extended across domains, as a mechanism for service chaining. A fair question for comparison might be, "If the same amount of research effort were put into this approach as has gone into fine-grained forwarding, which alternative would look better?"

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable feedback. They also thank Vyas Sekar, Mina Arashloo, Bharath Balasubramanian, Jennifer Gossels, Rob Harrison, Yaron Koral, Robert MacDavid, and Shankaranarayanan Narayanan for their feedback on earlier drafts.

REFERENCES

[1] P. Zave and J. Rexford, "The design space of network mobility," in *Recent Advances in Networking*, O. Bonaventure and H. Haddadi, Eds. New York, NY, USA: ACM SIGCOMM, 2013.

[2] D. A. Joseph, A. Tavakoli, and I. Stoica, "A policy-aware switching layer for data centers," in *Proc. ACM SIGCOMM Conf. Data Commun. - SIGCOMM*, 2008, pp. 51–62.

[3] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying middlebox policy enforcement using SDN," in *Proc. ACM SIGCOMM Conf. SIGCOMM - SIGCOMM*, 2013, pp. 27–38.

[4] Y. Zhang *et al.*, "StEERING: A software-defined networking for inline service chaining," in *Proc. 21st IEEE Int. Conf. Netw. Protocols (ICNP)*, Oct. 2013, pp. 1–10.

[5] X. Jin, L. E. Li, L. Vanbever, and J. Rexford, "SoftCell: Scalable and flexible cellular core network architecture," in *Proc. 9th ACM Conf. Emerg. Netw. Exp. Technol. - CoNEXT*, 2013, pp. 163–174.

[6] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing network-wide policies in the presence of dynamic Middlebox actions using FlowTags," in *Proc. USENIX Conf. Netw. Syst. Design Implement.*, Apr. 2014, pp. 533–546.

[7] A. Gember *et al.*, "Stratos: A network-aware orchestration layer for virtual middleboxes in clouds," 2013, *arXiv:1305.0209*. [Online]. Available: http://arxiv.org/abs/1305.0209

[8] B. Anwer, T. Benson, N. Feamster, and D. Levin, "Programming slick network functions," in *Proc. 1st ACM SIGCOMM Symp. Softw. Defined Netw. Res. SOSR*, 2015, p. 14.

[9] Z. A. Qazi, P. K. Penumarthi, V. Sekar, V. Gopalakrishnan, K. Joshi, and S. R. Das, "KLEIN: A minimally disruptive design for an elastic cellular core," in *Proc. Symp. SDN Res. SOSR*, 2016, pp. 2:1–2:12.

[10] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making Middleboxes someone else's problem: Network processing as a cloud service," in *Proc. SIGCOMM*, 2012, pp. 13–24.

[11] (Sep. 2015). *Faster DDoS Mitigation With Increased Customer Control: Introducing Verisign OpenHybrid Customer Activated Mitigation*. [Online]. Available: http://blogs.verisign.com/blog/entry/faster_ddos_mitigation_with_increa%sed

[12] D. Naylor *et al.*, "Multi-context TLS (mcTLS): Enabling secure in-network functionality in TLS," in *Proc. ACM Conf. Special Interest Group Data Commun. SIGCOMM*, 2015, pp. 199–212.

[13] C. Raiciu *et al.*, "How hard can it be? Designing and implementing a deployable multipath TCP," in *Proc. USENIX Conf. Networked Syst. Design Implement.*, 2012, pp. 399–412.

[14] (2019). *IETF Working Group on Service Function Chaining (SFC)*. [Online]. Available: http://datatracker.ietf.org/wg/sfc/

[15] (2019). *Contrail Feature Guide, Release 2.20, Service Chaining*. [Online]. Available: http://www.juniper.net/techpubs/en_US/contrail2.2/topics/task/configura%tion/service-chaining-vnc.html

[16] (2019). *Neutron Service Insertion and Chaining*. [Online]. Available: http://wiki.openstack.org/wiki/Neutron/ServiceInsertionAndChaining

[17] M. Arye, E. Nordstrom, R. Kiefer, J. Rexford, and M. J. Freedman, "A formally-verified migration protocol for mobile, multi-homed hosts," in *Proc. 20th IEEE Int. Conf. Netw. Protocols (ICNP)*, Oct. 2012, pp. 1–12.

[18] A. C. Snoeren and H. Balakrishnan, "An end-to-end approach to host mobility," in *Proc. 6th Annu. Int. Conf. Mobile Comput. Netw. - MobiCom*, 2000, pp. 155–166.

[19] E. Nordström *et al.*, "Serval: An end-host stack for service-centric networking," in *USENIX Conf. Networked Syst. Design Implement.*, Apr. 2012, pp. 85–98.

[20] P. Nikander, A. Gurtov, and T. R. Henderson, "Host identity protocol (HIP): Connectivity, mobility, multi-homing, security, and privacy over IPv4 and IPv6 networks," *IEEE Commun. Surveys Tuts.*, vol. 12, no. 2, pp. 186–204, 2nd Quart., 2010.

[21] R. Atkinson, S. Bhatti, and S. Hailes, "Evolving the Internet architecture through naming," *IEEE J. Sel. Areas Commun.*, vol. 28, no. 8, pp. 1319–1325, Oct. 2010.

[22] A. Rodríguez Natal *et al.*, "LISP-MN: Mobile networking through LISP," *Wireless Pers. Commun.*, vol. 70, no. 1, pp. 253–266, May 2013.

[23] C. Perkins, D. Johnson, and J. Arkko, *Mobility Support in IPv6*, document IETF Request for Comments 6275, Jul. 2011.

[24] C. Paasch and O. Bonaventure, "Multipath TCP," *Commun. ACM*, vol. 57, no. 4, pp. 51–57, Apr. 2014.

[25] J. R. Iyengar, P. D. Amer, and R. Stewart, "Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths," *IEEE/ACM Trans. Netw.*, vol. 14, no. 5, pp. 951–964, Oct. 2006.

[26] R. Krishnan, A. Ghanwani, J. Halpern, S. Kini, and D. Lopez, *SFC Long-Lived Flow Use Cases*, document IETF Internet Draft, draft-ietf-sfc-long-lived-flow-use-cases-03, Feb. 2015.

[27] P. Patel *et al.*, "Ananta: Cloud scale load balancing," in *Proc. SIGCOMM*, 2013, pp. 207–218.

[28] W. Haeffner, J. Napper, M. Stiemerling, D. Lopez, and J. Uttaro, Service Function Chaining Use Cases in Mobile Networks, document IETF Internet Draft, draft-ietf-sfc-use-case-mobility-09, Jan. 2019.

[29] A. Gember-Jacobson *et al.*, "OpenNF: Enabling Innovation in Network Function Control," in *Proc. SIGCOMM*, 2014, pp. 163–174.

[30] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes," in *Proc. 10th USENIX Symp. Netw. Syst. Design Implement.*, Lombard, IL, USA, Apr. 2013, pp. 227–240.

[31] G. J. Holzmann, *The Spin Model Checker: Primer Reference Manual*, 1st ed. Reading, MA, USA: Addison-Wesley, 2003.

[32] (2017). *Docker*. [Online]. Available: https://www.docker.com/

[33] (2019). *Mininet: An Instant Virtual Network on Your Laptop (or other PC)*. [Online]. Available: http://mininet.org/

[34] P. Zave, R. A. Ferreira, X. K. Zou, M. Morimoto, and J. Rexford, "Dynamic service chaining with dysco," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 57–70.

[35] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "SoftNIC: A Software NIC to Augment Hardware," EECS Dept., Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2015-155, May 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html

[36] Linux Foundation. (2019). *Data Plane Development Kit (DPDK)*. [Online]. Available: http://www.dpdk.org

[37] M. Bellare, R. Canetti, and H. Krawczyk, "Keying Hash Functions for Message Authentication," in *Proc. Annu. Int. Cryptol. Conf.*, London, UK: Springer-Verlag, 1996, pp. 1–15.

[38] D. Lebrun and O. Bonaventure, "Implementing IPv6 segment routing in the linux kernel," in *Proc. Appl. Netw. Res. Workshop - ANRW*, 2017, pp. 35–41.

[39] (2019). *Dysco Supplemental Material*. [Online]. Available: https://github.com/dysco/

[40] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000.

[41] (2019). *NGINX: A High-Performance HTTP Server and Reverse Proxy*. [Online]. Available: https://nginx.com/

[42] (2019). *Linux Netfilter*. [Online]. Available: http://www.netfilter.org

[43] (2019). *Linux TC*. [Online]. Available: http://lartc.org/manpages/tc.txt

[44] (2019). *PRADS*. [Online]. Available: http://gamelinux.github.io/prads/

[45] (2019). *The Bro Network Security Monitor*. [Online]. Available: https://www.zeek.org

[46] (2019). *Snort*. [Online]. Available: https://www.snort.org/

[47] (2019). *Suricata*. [Online]. Available: http://www.suricata-ids.org/

[48] (2019). *HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer*. [Online]. Available: http://www.haproxy.org/

[49] (2019). *Squid*. [Online]. Available: http://www.squid-cache.org/Intro/

[50] Y. Wang, S. Gobriel, R. Wang, T.-Y.-C. Tai, and C. Dumitrescu, "Hash table design and optimization for software virtual switches," in *Proc. Afternoon Workshop Kernel Bypassing Netw. - KBNets*, 2018, pp. 22–28.

[51] (2019). *Wrk: A HTTP Benchmarking Tool*. [Online]. Available: https://github.com/wg/wrk/

[52] A. Gember-Jacobson, A. Akella, R. Mahajan, and H. H. Liu, "Automatically repairing network control planes using an abstract representation," in *Proc. 26th Symp. Operating Syst. Princ.*, Oct. 2017, pp. 359–373.

[53] (2015). *Arbor Networks SP Solution*. [Online]. Available: http://www.arbornetworks.com/images/documents/Data%20Sheets/DS_SP_EN.p%df

[54] S. Palkar *et al.*, "E2: A framework for NFV applications," in *Proc. 25th Symp. Operating Syst. Princ. - SOSP*, 2015, pp. 121–136.

[55] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker, "Middleboxes no longer considered harmful," in *Proc. USENIX Symp. Operating Syst. Design Implement.*, 2004, p. 15.

[56] S. Guha and P. Francis, "An end-middle-end approach to connection establishment," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun. - SIGCOMM*, 2007, pp. 193–204.

[57] C. Nicutar, C. Paasch, M. Bagnulo, and C. Raiciu, "Evolving the Internet with connection acrobatics," in *Proc. workshop Hot Topics Middleboxes Netw. Function virtualization - HotMiddlebox*, 2013, pp. 7–12.

[58] A. Silvestro, N. Mohan, J. Kangasharju, F. Schneider, and X. Fu, "MUTE: MUlti-tier edge networks," in *Proc. 5th Workshop CrossCloud Infrastruct. Platforms - CrossCloud*, 2018, pp. 1–6.

[59] A. Langley *et al.*, "The QUIC transport protocol: Design and Internet-scale deployment," in *Proc. SIGCOMM*, 2017, pp. 183–193.

[60] A. Yadav and A. Venkataramani, "Msocket: System support for mobile, multipath, and middlebox-agnostic applications," in *Proc. IEEE 24th Int. Conf. Netw. Protocols (ICNP)*, Nov. 2016, pp. 1–10.

[61] G. Liu, Y. Ren, M. Yurchenko, K. K. Ramakrishnan, and T. Wood, "Microboxes: High performance NFV with customizable, asynchronous TCP stacks and dynamic subscriptions," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 504–517.

[62] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park, "mOS: A reusable networking stack for flow monitoring middleboxes," in *Proc. NSDI*, 2017, pp. 113–129.

Pamela Zave received the A.B. degree in English from Cornell University and the Ph.D. degree in computer sciences from the University of Wisconsin–Madison. She has held positions at the University of Maryland, Bell Laboratories, and AT&T Laboratories. She is currently a Research Associate with Princeton University. She also holds 31 patents. She is an ACM Fellow, an AT&T Fellow, and the 2017 winner of the IEEE Computer Society's Harlan D. Mills Award for sustained contributions to software engineering research and practice. Her work on the foundations of requirements engineering has been recognized with three Ten-Year Most Influential Paper awards. At AT&T, she led a group that developed two successful large-scale telecom systems based on her Distributed Feature Composition architecture, including AT&T's first public voice-over-IP offering.

Fabrício B. Carvalho received the B.Sc. and M.S. degrees from UFMS in 2011 and 2014, respectively, where he is currently pursuing the Ph.D. degree. He is also a Lecturer with UFMT. His research interests include computer networks and distributed systems.

Ronaldo A. Ferreira received the B.Sc. degree from UFMS in 1992, the M.S. degree from the University of Campinas in 1998, and the Ph.D. degree from Purdue University in 2006, all in computer science. He was a Visiting Research Scholar and a Visiting Associate Professor with Princeton University from 2014 to 2016. He served as the Chair for the Special Interest Group on Computer Networks and Distributed Systems of the Brazilian Computing Society (SBC) from 2011 to 2013. He is currently an Associate Professor of Computer Science in the College of Computing at UFMS. His research interests include computer networks and distributed systems.

Jennifer Rexford (Fellow, IEEE) received the B.S.E. degree in electrical engineering from Princeton University in 1991, and the Ph.D. degree in electrical engineering and computer science from the University of Michigan in 1996. She is currently the Gordon Y. S. Wu Professor of Engineering and the Chair of Computer Science with Princeton University. Before joining Princeton in 2005, she worked for eight years at AT&T Labs–Research. She is a coauthor of the book *Web Protocols and Practice* (Addison-Wesley, May 2001). She is an ACM Fellow in 2008 and a member of the American Academy of Arts and Sciences in 2013 and the National Academy of Engineering in 2014. She was the 2004 winner of ACM's Grace Murray Hopper Award for outstanding young computer professional, the ACM Athena Lecturer Award in 2016, the NCWIT Harrold and Notkin Research and Graduate Mentoring Award in 2017, the ACM SIGCOMM Award for lifetime contributions in 2018, and the IEEE Internet Award in 2019. She served as the Chair for ACM SIGCOMM from 2003 to 2007.

Masaharu Morimoto received the B.E. and M.E. degrees from Kobe University in 2004 and 2006, respectively. In 2006, he joined NEC Corporation, Japan, as a Research Engineer in computer networks. He worked as a Visiting Researcher with Princeton University in 2015, and worked with NEC Corporation of America in 2016. Since 2019, he has been the Head of the Intelligent Sensing Group, NEC Laboratories Singapore. He is currently a Principal Researcher with NEC Corporation.

Xuan Kelvin Zou received the B.Sc. degree from the University of Illinois at Urbana–Champaign in 2013 and the M.A. degree from Princeton University in 2015, all in computer science. He is currently a Staff Engineer with ByteDance Inc. His research interests include computer network and systems, and applied machine learning infrastructure.