Dynamic Service Chaining with Dysco

Pamela Zave[1], Ronaldo A. Ferreira[2], X. Kelvin Zou[3], Masaharu Morimoto[4], and Jennifer Rexford[5]

[1]AT&T Labs—Research

[2]College of Computing, Federal University of Mato Grosso do Sul

[3]Google

[4]NEC Corporation of America

[5]Department of Computer Science, Princeton University

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 the switches—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." The situation is even more challenging if the sequence of middleboxes (the "service chain") needs to *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. 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. 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 Linux-based prototype, we show that Dysco is provably correct, highly scalable, and able to reconfigure service chains across a range of middleboxes.

1. 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 deserves as much re-examination of approaches as mobility and multihoming.

Most existing research proposals use a logically centralized controller to install fine-grained forwarding rules in the switches, to steer traffic through the right sequence of middleboxes [1–8]. Even for the simplest service chains, these solutions have several limitations:

- They require special equipment (e.g., SDN switches) that can perform match-action processing based on multiple packet header fields.
- They rely on real-time response from the central controller to handle frequent events, including link failures, and the addition of new middlebox instances.
- They need switch-level state that grows with the diversity 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 policy changes, topology changes, or fluctuating loads must take care to ensure that all packets of a session traverse the same middleboxes.
- Outsourcing middleboxes to the cloud [9] or other third-party providers [10] becomes difficult, since the controller cannot control the end-to-end path.

Further, there are common situations in which fine-grained forwarding rules are insufficient. These include:

- Some middleboxes (e.g., NATs) modify the "five-tuple" of packets in unpredictable ways, so that packets emerging from the middlebox cannot be associated reliably with the packets that went in.
- Some middleboxes classify packets to choose which middlebox should come next. These middleboxes must be able to affect forwarding of their outgoing packets.
- 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) [11]. Regardless of access network, all these paths must converge at the same middlebox.

Fine-grained forwarding is even more limiting when the sequence of middleboxes for an ongoing session needs

to change. We call changing middleboxes mid-session *dynamic reconfiguration,* and it is useful in a wide range of situations (see also [12]):

- After directing a request to a backend server, a load balancer can remove itself from the path and, hence, avoid being a single point of failure.
- An intrusion detection system, Web proxy cache, or ad-inserting proxy can remove itself after its work for a session is done.
- When coarse-grained traffic measurements (e.g., Netflow data) identify suspicious traffic, it can be directed through a packet scrubber for further analysis.
- When the network is congested, video sessions can be directed through compression middleboxes [13].
- An overloaded middlebox, or one that is undergoing maintenance, can be replaced with another middlebox of the same type (e.g., see [14, 15]).
- When an end-host moves to a new location, a middlebox can be added temporarily to buffer and redirect traffic from the old location. In addition, the old middleboxes in the service chain can be replaced with new ones closer to the new location.

Note that removing a middlebox goes beyond simply pushing packet forwarding into the kernel, and removes the machine from the path entirely. This reduces latency and increases reliability, while conserving middlebox resources for sessions that actually need them.

In response to these difficulties with fine-grained forwarding, emerging industry solutions encapsulate packets so that their *destination addresses alone* cause them to be forwarded through the service chain [16–18]. This is a step in the right direction, but does not go far enough. A *session protocol* can also manipulate destination addresses for basic service chaining, while using signaling to avoid encapsulation and achieve dynamic reconfiguration. Thus a session protocol can overcome all the limitations listed above, and make service chaining independent of routing. In the spirit of the end-to-end argument, all of the key functions are performed by *hosts*—whether end-hosts or middlebox hosts. Session protocols already provide effective and efficient support for mobility [19–25] and multihoming [26, 27], and we complete the exploration of this "design pattern" by focusing on middleboxes. We introduce Dysco, a session protocol for steering packets through middleboxes and reconfiguring established service chains.

The paper makes the following contributions:

Compatibility with TCP: Dysco is designed to be compatible with TCP so that it requires 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. Although the Dysco approach should be compatible with most transport protocols, we decided that grappling with the complexities of TCP was more important than demonstrating generality in this dimension.

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. To ensure that distributed control is exercised safely, Dysco manages the contention when Dysco agents for different middleboxes attempt to reconfigure overlapping segments of the same session at the same time.

Generalized dynamic reconfiguration: 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. The middlebox might also have changed the size of a byte stream (e.g., by transcoding or adding/removing content). Packet buffering is usually not necessary, but if used to freeze server state for migration or to preserve packet order, it can be performed exclusively by hosts.

Protocol verification: We have a formal model of the Dysco protocol that is detailed enough to manipulate sequence numbers, yet abstract enough to be verified by means of model-checking. 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 includes a Linux kernel module that intercepts packets in the network device, so it works with unmodified applications and a wide range of middleboxes. The kernel module supports Linux namespaces, which makes it suitable for virtualized environments (e.g., Docker [28]) and experimentation with Mininet [29]. Experiments show that session setup is fast, steady-state throughput is high, and disruption during reconfiguration is small.

Although there is work yet to do on Dysco, the results in this paper show that the approach can work in practice. The most exciting future prospect is to integrate Dysco with session-based approaches to mobility and multihoming. While the mainstream solutions for handling middleboxes, mobility, and multihoming rely on routing and fine-grained forwarding rules, we believe that an integrated session-based approach is worth exploring because implementing functions in the hosts offers much better flexibility and scalability.

2. 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 sec-

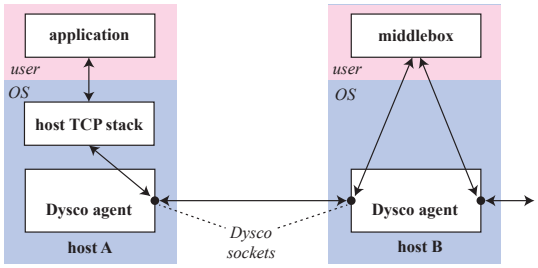


Figure 2: Data flow inside hosts with Dysco agents.

tion, we introduce the Dysco architecture and give an overview of the protocol; in §3, we expand on how Dysco can reconfigure an existing service chain.

2.1 Concepts: Subsessions and service chain

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. A service chain is set up when the TCP session is set up. The service chain sometimes has the same endpoints as the TCP session, as shown in Figure 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 usual, congestion control and retransmission are performed end-to-end (see Figure 2). At the same time, 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.

More generally, a service chain may span 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 have two TCP sessions. The Dysco agent can simply present data to the middlebox application with the TCP session identifier that applies at that point in the service chain. At some point, the proxy's work may be done, 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 (e.g., a "splice" call to relegate further handling of the traffic to the kernel). After deleting a session-terminating proxy, the resulting service chain would correspond to a single TCP session.

Likewise, a TCP session may span one or more service chains, particularly under *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. Thus a TCP session can have one or more independent service chains, enabling partial deployment.

2.2 Components: Agents and policy server

The Dysco agents implement the Dysco protocol (to create, reconfigure, and tear down service chains) and rewrite packet headers (to present the right identifiers to applications, middleboxes, and other agents). A policy server determines which traffic should traverse which kinds of middleboxes, and can optionally trigger reconfiguration of groups of service chains. Compared to an SDN controller, the policy server need not control traffic-steering decisions for individual sessions or install any switch-level rules.

Identifying the service chain: The first Dysco agent in a service chain gets the policy for the chain (see §2.3). Yet, the policy server need not be involved with individual sessions. For example, initial policies can be pre-loaded or cached, so that an agent does not have to query a policy server for every new session. 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, including resolving any contention if multiple portions of a service chain try to change at the same time.

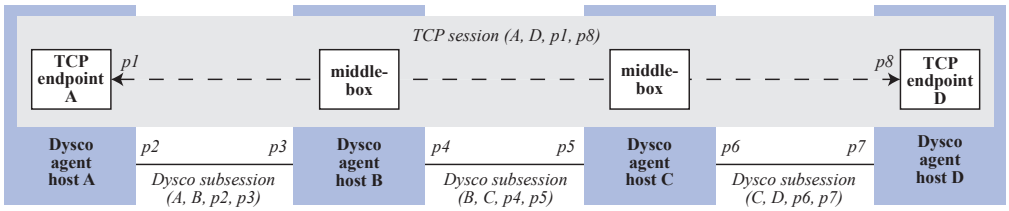


Figure 1: A TCP session with its Dysco subsessions.

2.3 Phases of the protocol

Establishing a service chain: An initial service chain is established during TCP session setup. Using the example of Figure 1, the Dysco agent at host A intercepts the outbound SYN packet. 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 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 endpoint.

Middleboxes that modify the five-tuple: If such a middlebox, e.g., a NAT, has a Dysco agent, the 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 [5] and Stratos [6], because they are applied *only* to SYN packets, are *never* sent to the network, and are meaningful only to the agent that inserts and removes them.) A middlebox that modifies the five-tuple 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.

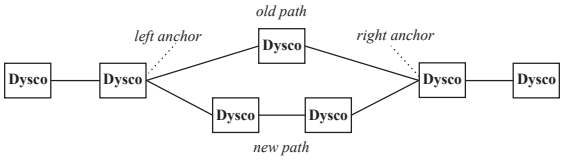


Figure 3: Agents reconfigure a segment of a session, replacing an old path with one middlebox by a new path with two.

Agents reconfiguring a segment of a session: Reconfiguration of a service chain can be triggered by the policy server or the middleboxes themselves, but it is always initiated by a Dysco agent and carried out exclusively by the agents in the chain. Reconfiguration operates on a *segment*, consisting of some contiguous subsessions and the associated hosts. As shown in Figure 3, the agents at the two unvarying ends of a segment are the *left anchor* and *right anchor*. An anchor can be the agent for a middlebox or end-host. If the old path consists of a single subsession (with no middleboxes), and the new path has at least one middlebox, then middleboxes have been *inserted*. Reverse old and new above, and middleboxes have been *deleted*. If both old and new paths have middleboxes, then the old ones have been *replaced* by the new. The anchors cooperate through control signaling to replace the old path of the segment with a new path. Reconfiguration is always initiated by the left anchor, which must know the address of the right anchor and the list of middlebox addresses to be inserted in the new path (if any). There is no need for packet buffering, because new data can always be sent on one of the two paths.

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 time out 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 heartbeat signals to keep good subsessions alive.

Security: Like other session protocols [19–24], Dysco is vulnerable to adversaries that inject or modify control messages. Dysco can adopt the same solutions to protect against both off-path attacks (e.g., an initial exchange of nonces, with nonces included in all control messages) and on-path attacks (e.g., encrypting control messages within a chain and with the policy server). The agents of a service chain are cooperating entities that must trust each other. Excluding untrusted hosts from a service chain is straightforward, since a service chain can span just a portion of a TCP session. Cooperating domains can exchange information about trusted middlebox hosts (by IP address and optional public key) so a middlebox in one domain can establish a subsession with a trusted middlebox in another.

3. DYNAMIC RECONFIGURATION

3.1 Protocol overview

To reconfigure a service chain Dysco agents use control packets, each carrying in its body the associated session identifier. Reconfiguration is always initiated by the Dysco agent acting as the left anchor, as in Figure 3. Although reconfiguration can be triggered by a controller or other middlebox, the triggering component must always communicate with the left anchor to request it to execute the protocol.

Just as the Dysco agent for A in Figure 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, . . . , rightAnchor]* with the middleboxes and right anchor of the new path that will replace the old path. Typically the list comes from the triggering agent. If a middlebox wants to delete itself, it sends a triggering message to the agent on its left with the address list *[myRightNeighbor]*, so the left anchor has an address list containing only a right anchor.

Figure 4 shows the control packets exchanged by the anchors during the first phase of a simple, successful reconfiguration. The red packets travel on the old path, so they are forwarded through the Dysco agents of current middleboxes (the *delta* fields will be explained in §3.3). The blue three-way SYN handshake sets up the new path within the service chain. As in §2, the SYN carries an address list so that the Dysco agents can include all the addressed middleboxes before the right anchor. During this phase normal data transmission continues

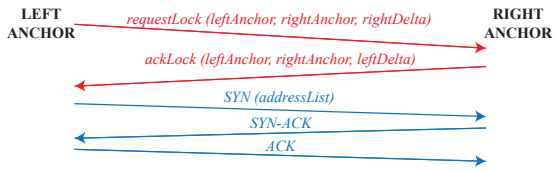


Figure 4: Control packets exchanged for reconfiguration. Red packets travel on the old path, blue on the new path.

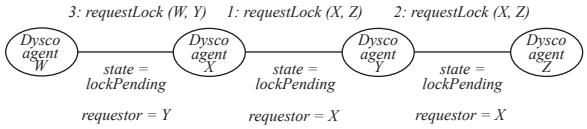


Figure 5: Contention to reconfigure overlapping segments.

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 anchors tear down the old path and discard the state kept for it. In subsequent sections we provide protocol details, organized by significant issues and challenges.

3.2 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. Figure 5 shows how the protocol prevents this.

For each subsession, the agent on its left maintains a state that is one of *unlocked, lockPending,* or *locked.* If it is *lockPending* or *locked*, then variable *requestor* holds the left anchor of the request for which it is pending or locked. If an agent receives *requestLock(leftAnchor, rightAnchor)* from the left, 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 *requestor* to *leftAnchor.* If there is no contention, the same agent will receive a matching *ackLock* from the right. It will forward the *ackLock* and set the subsession state to *locked.* In the figure, a request to lock the segment from X to Z has propagated from X to Z (packets 1 and 2).

Meanwhile agent W might be triggered to lock the segment from W to Y. Its request (packet 3) will be blocked at X because the subsession to its right is *lockPending.* Eventually X will receive either *ackLock* or

nackLock in response to its own request. If *ackLock*, it replies with *nackLock* to the request from W. As a *nackLock* propagates leftward, *lockPending* states are reset to *unlocked*. On the other hand, if X receives *nackLock* in response to its own request, then the subsession to its right becomes unlocked, and it can forward the saved request from W.

3.3 Sequence-number deltas

Some middleboxes increase or decrease the size of a byte stream (by transcoding, inserting or deleting content, etc.). 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 the reverse direction with a different sequence number than the end-host. If a middlebox with a delta is deleted, the discrepancy in sequence numbers must be fixed somewhere else.

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, for data coming in on the new path or going out on the new path, the anchor must apply its delta to packets. The table shows how. To simplify the presentation, we assume that sequence numbers do not wrap around to zero.

packet direction	how apply delta	to which field
in	add	sequence number
out	subtract	acknowledgment number

3.4 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. 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. The anchors send termination messages on the old path, then clean up the extra state variables.

If a stateful middlebox in the session is being replaced, then additional synchronization is needed. 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 [15]. 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

[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.

anchors must buffer incoming data. The anchors can also buffer data and clear the old path before using the new path if other middleboxes (outside the reconfigured segment) cannot tolerate re-ordered packets.

3.5 Failures

If control packets are lost, then the protocol detects this and retransmits them. 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 abort the reconfiguration, so the session continues to use the old path. After this the subsessions of the old path between the anchors are still *locked*, so that they cannot be reconfigured in the future. So the left anchor sends a *cancelLock* control packet to the right anchor, the right anchor replies with an *ackCancel*, and all the agents that receive these signals unlock their subsessions.

3.6 Properties and verification

While the Dysco protocol presented here is quite simple, the design process required grappling with a large number of functions, cases, and potential race conditions (e.g., arrival of a FIN in either direction at any time during reconfiguration). The iterative design of our protocol was possible because we modeled the protocol in the Promela language, and exhaustively checked its behavior with the Spin model-checker [30]. At each stage of the design process, we received immediate feedback on bugs and unresolved issues.

In Promela each Dysco agent is implemented with nondeterministic concurrent processes communicating through message queues. The model can be configured with up to two middleboxes in an initial session. It is also configured with one or more reconfigurations that can be attempted. Some of the nondeterminism in the model reflects choices that can be made by application code, but most nondeterminism is due to the fact that concurrent events can be interleaved in all possible ways. In other words, the execution traces of the model cover all possible network delays and scheduling decisions. In a typical run for a typical configuration, Spin constructs a global state machine of all possible behaviors with 100 million state transitions. The model, along with extensive documentation of design, modeling abstractions, and Spin runs, can be found at [31].

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 endpoints are correct.
- All sessions terminate cleanly.

No doubt there will be future versions of the Dysco protocol with enhancements and optimizations, which we will continue to track with our model and verify. Ultimately we need only one implementation of the protocol, and it will reach robustness much faster than usual because of extensive modeling and verification.

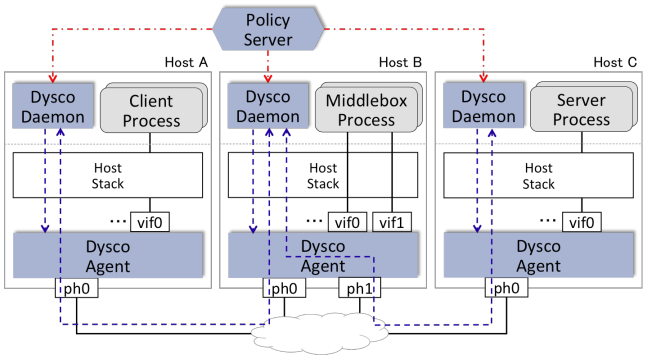


Figure 6: Implementation, where solid black lines represent the data path, blue dashed lines the control path, and red dashed-dotted lines the management path used to distribute policies.

4. DYSCO PROTOTYPE

Our Dysco prototype consists of a kernel-level agent that communicates with an external policy server through a user-space daemon, as shown in Figure 6.

4.1 Dysco components and interfaces

Agent: The Dysco agent supports unmodified end-host applications, middleboxes, and host network stacks by intercepting packets going to/from the network. In our prototype, the Dysco agent is a Linux kernel module that intercepts packets in the device driver. Even though the Linux kernel is not the fastest option for high-performance middleboxes, we decided to do an in-kernel implementation to transparently support TCP-terminating applications (e.g., proxies, HTTP servers, and clients), middleboxes that use `libpcap` to get/send packets from/to the network (e.g., Bro [32], Snort [33][2], and PRADS [34]), and middleboxes that run in the Linux kernel (e.g., Traffic Controller (tc) [35] and Iptables/Netfilter Firewall [36]). Our prototype also supports network namespaces for virtualized environments, such as Docker and Mininet. The kernel module con-

[2]The latest versions of Snort use a Data Acquisition Layer (DAQ) that allows the use of different packet acquisition methods, such as `libpcap` and DPDK.

sists of over 6000 lines of C code, and adds only 16 lines of C code in the device drivers to call the functions that intercept the packets and initialize and remove the module. Our prototype currently supports the Intel ixgbe driver (for 10 gigabit NICs), the e1000 driver (for 1 gigabit NICs), veth (for virtual interfaces), and the Linux bridge.

Middleboxes: Dysco supports unmodified middlebox applications, and we have successfully run with NGINX [40], HAProxy [38], Iptables/Netfilter [36], and Linux tc [35]. Most middleboxes send and receive data via `libpcap`, user socket, or Linux `sk_buff`. The application can be (i) *passive* (e.g., PRADS [34], Bro [32], Snort [33], Suricata [37], Linux tc [35], Iptables/Netfilter Firewall [36]) or (ii) *active* if it changes the TCP session identifier or sequence numbers (e.g., Iptables/Netfilter NAT [36], HAProxy [38], Squid [39]). Passive middleboxes that use `libpcap` or `sk_buff` run transparently and unmodified with Dysco. To support the removal of TCP-terminating proxies, the Dysco agent intercepts the Linux "splice" system call and then invokes the reconfiguration protocol. We also support a `dysco_splice` system call that a (modified) middlebox can use to trigger its own removal. We discuss these in more detail below. 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 [14].

Daemon: The Dysco agent performs session setup and teardown, as well as data transfers, directly in the kernel. We implemented the reconfiguration protocol in a separate user-space daemon for ease of implementation and debugging. Reconfiguration messages are infrequent, compared to data packets, so the small performance penalty for handling reconfiguration in user space is acceptable. The daemon communicates with the Dysco agent in the kernel via `netlink` (a native Linux IPC function), with other Dysco agents via UDP, and with the policy server via TCP. Our prototype includes a library for a simple management protocol for the daemon and the policy server. The daemon compiles and forwards to the kernel the policies received from the policy server, triggers reconfiguration, and performs state migration when replacing one middlebox with another (by importing and exporting state, and serializing and sending the state to another middlebox).

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 daemons. Commands can be batched and distributed to different hosts using shell scripts. The policy server and the Dysco daemon consist of over 5000 lines of Go of which 3000 lines are a shared library for message serialization and reliable UDP transmission. The source code of Dysco as well as the shell scripts used for the evaluation are available at [31].

4.2 Protocol details

Tagging SYN packets: The local tags added to SYN packets, as described in §2.3, 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 number 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.

Minimizing contention during lookups: The agent stores the mapping between incoming and outgoing five-tuples in a hash table that uses RCU (Read-Copy-Update) locks for minimizing contention during lookups. Since entries are added to the hash table for each new TCP session, a naïve locking strategy based on mutexes or spin locks would degrade the performance significantly. To support Linux namespaces, the agent maintains one translation table per namespace.

UDP messages for reconfiguration: Our daemon implements the reconfiguration protocol using UDP messages. We chose to use UDP in user space to facilitate development and debugging. Also, reconfigurations do not occur frequently, so the performance requirements are not as stringent as for the data plane. The UDP control messages, described in §3, carry the five-tuples of the TCP sessions going through the reconfiguration, so the Dysco daemon and agent can associate the control message with the session state inside the kernel.

Beyond the protocol outlined in §3, we now address several interoperability issues that arise for middleboxes that terminate TCP sessions, including layer-7 load balancers and proxies.

Triggering a reconfiguration using "splice": To deal with middleboxes that terminate TCP sessions and want to remove themselves, Dysco offers two options First, we have 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:

```
int dysco_splice(int fd_in, int fd_out, int delta)
```

This option requires the modification of a middlebox to call the library function. Second, we support unmodified middleboxes that use the Linux's "splice" system call. For this case, we provide a shared library that in-

tercepts the C library functions used for network communication (e.g., socket, accept, connect, splice, close, and the read and write functions). The shared library must be preloaded using LD_PRELOAD. Each function of the shared library first calls the original function from the C library and then records the result of the operation. For example, a function that intercepts any of the read calls records the amount of data read from a socket. When the splice function is called, the shared library uses the recorded information to compute the delta between two sockets and find the information about the TCP sessions associated with the sockets (i.e., the five-tuples of the two TCP sessions). Note that the Linux splice call receives a socket and a pipe as parameters. The first call to splice just sets data structures internal to the kernel. The operation is performed only on the second call to splice. We track both calls and trigger a reconfiguration after the second call, when we have all the information needed.

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.

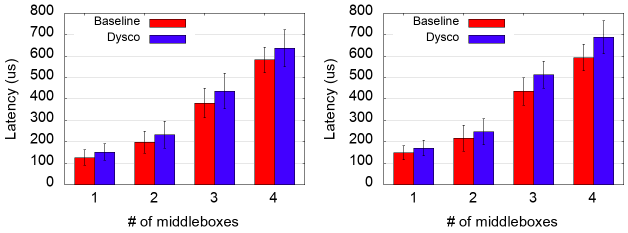
5. PERFORMANCE EVALUATION

We now evaluate Dysco in the three main phases of a session across different network settings. First, we measure the latencies for session initiation to quantify the overhead introduced by subsession setup and including middlebox address lists in the TCP SYN packets. Second, we measure the throughput of a session during normal data transfer to show that the Dysco agents can forward packets at very high speed. Third, we show that dynamic reconfiguration improves end-to-end performance and introduces minimal transient disruptions.

Our testbed consists of a NEC DX2000 blade server with 11 hosts, each with one Intel eight-core Xeon D 2.1 GHz processor, 64 GB of memory, and two 10Gbps NICs. The two NICs of each host are connected to two independent layer-two switches, forming two independent LANs. The eleven 11 hosts run Ubuntu Linux with kernel 4.4.0.

5.1 Session initiation

Figure 7 shows session setup latency under two scenarios: Dysco and middleboxes inserted by IP routing (Baseline). We do not run a middlebox application (i.e., the middleboxes simply forward packets in both directions), so we only measure the overhead of the Dysco protocol. The scenario with one middlebox has three hosts, and the one with four middleboxes has six hosts connected in a line. 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. Figure 7(a) shows the latencies when the checksum computation is offloaded to the NIC and Figure 7(b) when the computation is not offloaded. The worst case for Dysco is with four middleboxes, and when the checksum computation is not offloaded to the NIC. The time difference between the two averages, in this case, is only 94μ. The measured latencies are insignificant compared to the overhead for middlebox applications to transfer packets to user space to perform network functions and represent less than 0.5% in a TCP session with RTT of 20 ms in the worst case. From now on, we report results only for the cases where checksum and TCP segmentation are not offloaded to the NIC, as these represent the worst cases for Dysco.



(a) Checksum offloading **(b)** No checksum offloading

Figure 7: Latency for session initiation.

5.2 Data-plane throughput

Figure 8 shows the goodput, measured at the receivers, of multiple TCP sessions between four clients and four servers connected via a middlebox that sim-

ply forwards traffic between the clients and the servers. Again, we do not run an application on the middlebox to quantify just the Dysco overhead. The figure shows no noticeable difference between the performance of Dysco and the baseline case; the differences between the two cases are always within one standard deviation and are less than 1.5 percentage points in the worst case. We show the results up to 1000 sessions because from this point on the link becomes the bottleneck.

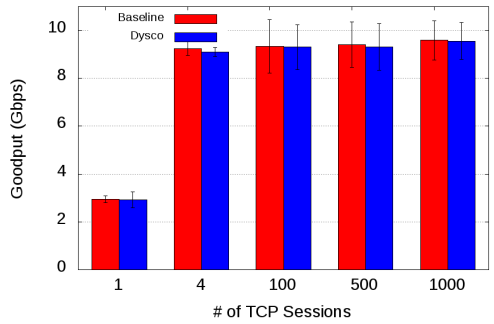


Figure 8: Goodput of Dysco compared with the baseline.

We also measured the number of requests that NGINX [40], a popular HTTP server, can sustain under Dysco and compared the results with the baseline. The measurement was performed with `wrk` [41], an HTTP benchmarking tool, with 16 threads and four hundred persistent connections, as recommended in [41]. NGINX is able to serve more than 300,000 connections per second when only one middlebox is between the client and the server, and a little under 300,000 connections per second when four middleboxes are between the client and the server. The results are consistent with the throughput measurements, and the largest difference between Dysco and the baseline is less than 1.8 percentage points.

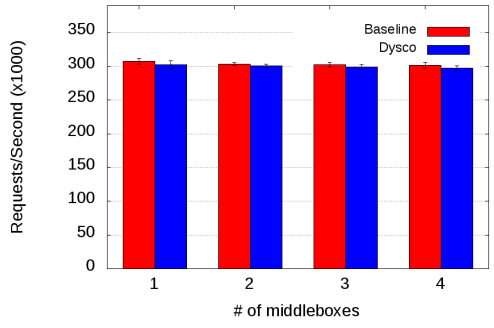


Figure 9: Number of HTTP requests per second NGINX can serve under Dysco and the baseline.

5.3 Dynamic reconfiguration

In this section, we investigate a few scenarios of dynamic reconfiguration. We use the logical topology of Figure 10; one of the hosts works as the router, and each IP subnet is on a different VLAN.

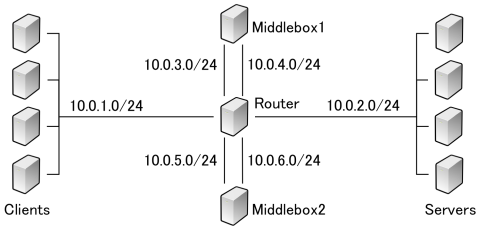


Figure 10: Testbed topology for the performance evaluation of the reconfiguration experiments.

Middlebox deletion: We run TCP sessions from four Clients to four Servers, passing through the Router and Middlebox1, which is running a TCP proxy. After 40, 60, 80, and 100 seconds, we trigger reconfigurations that remove Middlebox1 from a client-server pair and directs 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 150 TCP sessions for a total of 600 simultaneous sessions.

The top of Figure 11 shows the throughput 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 600 sessions no longer go through the proxy, the overall goodput has doubled from the time interval before the reconfigurations started. The bottom of Figure 11 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.

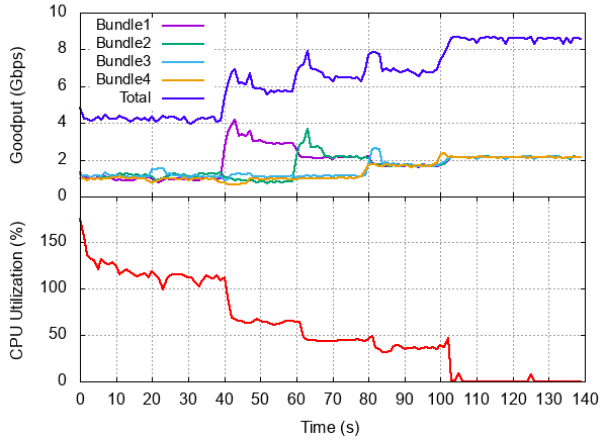


Figure 11: Goodput of TCP sessions (top) and CPU Utilization at the proxy (bottom) before and after multiple reconfigurations.

We can see in Figure 11 that the reconfigurations are

successful and the traffic reaches steady-state behavior after 100 seconds. Figure 12 shows that reconfiguration time is short: almost 80% of reconfigurations took less than 2ms and 98.7% less than 4ms. The few larger values happen when control messages are lost and need to be retransmitted.

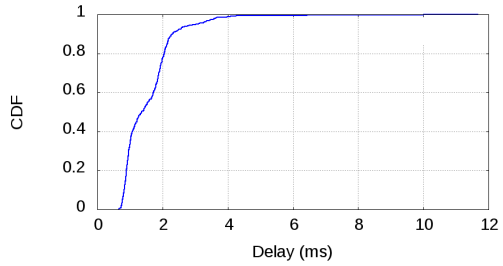
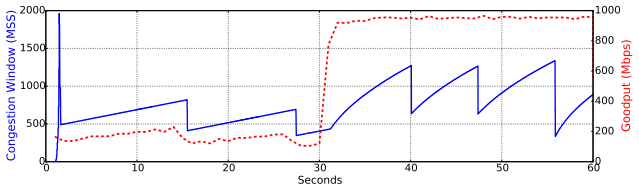
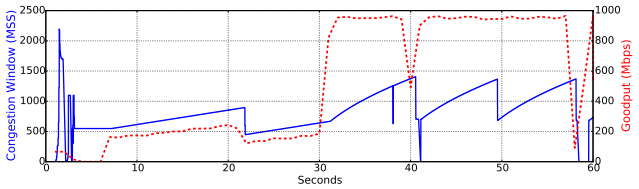


Figure 12: CDF of the reconfiguration time for proxy removal.



(a) TCP SACK enabled



(b) TCP SACK disabled

Figure 13: TCP performance during reconfiguration.

Session disruption: We investigate the transient performance of a session after removing a proxy, where the new path is faster than the old one (so packets may arrive out of order to the destination). To better control network latency, we simulate the testbed topology in Mininet, where we can introduce different link delays and bandwidths. Figure 13(a) plots the congestion window (left y-axis) and TCP goodput (right y-axis) during a proxy removal. The proxy triggers the reconfiguration 30 seconds after the beginning of the session. As we can see, the session experiences no disruption. Figure 13(b) shows why the Dysco agents must handle TCP options—with TCP SACK disabled, packet losses temporarily degrade session performance.

Middlebox replacement with state transfer: Middleboxes may need to transfer internal state as part of middlebox replacement, and ensure that the new component is ready before receiving its first packet [14, 42]. While routing solutions rely on clever synchronization of switches and a controller, Dysco uses simpler mechanisms, as the anchors can coordinate to determine when the new component is ready.

To experiment with state transfer, we extended the Dysco daemon to get state information of the Linux Netfilter firewall, serialize the data using JSON, and send the serialized data to another Dysco daemon. We did not modify Netfilter to interact with Dysco, so the interaction between Dysco and Netfilter is completely transparent to the firewall. The internal state of Netfilter can be obtained by running the `conntrack` Linux utility with a filter to select the relevant session(s).

The reconfiguration involves two Netfilter firewalls running on Middlebox1 and Middlebox2, and TCP sessions running from three clients to three servers in Figure 10. The client and the server are the left and right anchors, respectively. Upon receiving a SYN-ACK message (indicating that the new path is established), the left anchor sends a state-transfer message to Middlebox1 with the session that is migrating to the new path, the address of Middlebox2, and the addresses of the left and right anchors. The Dysco daemon running on Middlebox1 gets and sends the state information directly to the Dysco agent running on Middlebox2 and waits for a notification that the state is installed before notifying the left and right anchors that the new path is ready.

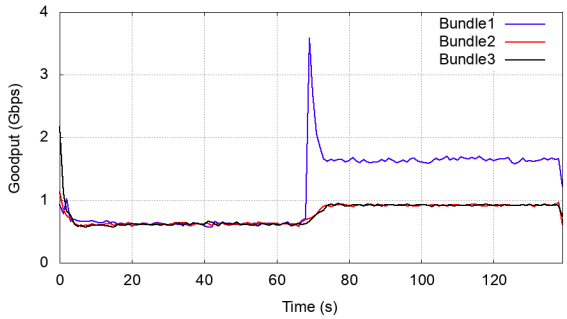


Figure 14: Goodput during reconfiguration and state migration.

Figure 14 shows the goodput of three bundles of 100 sessions running while the state from Middlebox1 is transferred to Middlebox2. For this experiment, the link speeds on the two middleboxes were limited to 2Gbps to avoid creating a bottleneck on the router of Figure 10. The time series represents the goodput measured at one-second intervals. The purple line is a bundle of sessions that runs through Middlebox1 until the 70s mark and then changes to a new path that goes through Middlebox2. After the purple-line session is moved from Middlebox1 to Middlebox2, the goodput of all sessions increases. The overall goodput of the sessions that now go through Middlebox2 is almost twice the goodput of the sessions that stayed on Middlebox1. The session that migrates from Middlebox1 to Middlebox2 does not suffer performance degradation (i.e., no

lost or reordered packets) and is not blocked by the firewall on Middlebox2. The average reconfiguration time for the 100 migrations, including state transfer and measured from the moment a SYN message is sent until the new path is used, was less than 100 ms. Comparing with the times from Figure 12, we can see that in this case the state transfer dominates the reconfiguration time.

6. RELATED WORK

BGP: Early solutions to dynamic service chaining manipulate BGP to "hijack" traffic, either within a single domain [43] or across the wide area [10]. However, manipulating BGP is risky in the wide area, and operates at the coarse level of destination IP prefixes rather than individual sessions. Plus, it is difficult to use BGP to insert multiple middleboxes in a service chain.

DOA: Like Dysco, DOA [44] uses a session protocol for middlebox traffic steering. 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 middleboxes inserted by administrators), and (iv) DOA uses encapsulation, so that both high- and low-level addresses are included in each packet. Extra addresses increase packet size, which may cause MTU problems.

NUTSS: In the NUTSS architecture [45], 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 so that later session packets can pass through middleboxes such as firewalls. NUTSS requires changes to all end-hosts and middleboxes.

mcTLS: Multi-context TLS (mcTLS) [46] enables middleboxes to operate on encrypted traffic, through a signaling protocol that (i) establishes a TCP connection for each hop in the service chain and (ii) exchanges the relevant security information for decrypting and reencrypting the data. Like Dysco, mcTLS has a list of middleboxes in a set-up message, albeit using the TLS Hello message, rather than the TCP SYN packet. However, mcTLS does not support dynamic service chaining and does not offer end-to-end TCP semantics (for connection set-up, retransmission, congestion control, etc.).

Multipath TCP: Nicutar et al. [47] 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 endpoint 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 endpoint from unwanted sessions as a firewall does, cannot choose the endpoint of a session as a load balancer does, and are not guaranteed to see all packets within a session.

OpenNF: OpenNF [14] (and also Split-Merge [15]) 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 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 all packet buffering.

Stratos and E2: Stratos [6] and E2 [48] are designed for middlebox deployment within clouds. They offer integrated solutions for managing middleboxes, including elastic scaling of middlebox instances, fault-tolerance, and placement, as well as (static) service chaining. They use fine-grained forwarding rules for traffic steering, inheriting the scaling challenges mentioned in §1. Dysco is much simpler and narrower in scope than these efforts; as such, it can be readily combined with various approaches to middlebox management, including these.

NSH: Network Service Header [16] is an encapsulation format for steering traffic through a service chain. Packets in this form cannot traverse NATs, and there is no mechanism for dynamic reconfiguration. Dysco also improves on encapsulation by using rewriting to prevent increases in packet size.

7. LIMITATIONS AND CONCLUSION

Our prototype does not implement the security mechanisms presented in §2.3. Our choice of UDP for control signaling has the unfortunate side-effect that reconfiguration does not currently work across an unmodified NAT; TCP packets on the new path from right to left anchors will appear unsolicited to a NAT between them, because the new path was set up with UDP. In addition, currently reconfiguration cannot be used to replace a failed middlebox, because the old path is needed by the protocol. We have strategies for adding security and replacing UDP with TCP signaling. Replacing a failed stateful middlebox is almost impossible, but we have a strategy for replacing failed stateless middleboxes.

The next steps for Dysco are to remove these limitations, make additional performance measurements, and deploy Dysco in a real network. In the longer term we will integrate Dysco with session-protocol approaches

to mobility and multihoming for a single, unified solution. In this integrated solution, mechanisms from contributing session protocols that now have overlapping purposes can be coalesced for greater simplicity.

8. REFERENCES

[1] D. A. Joseph, A. Tavakoli, and I. Stoica, "A policy-aware switching layer for data centers," in *ACM SIGCOMM*, Aug. 2008.

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

[3] Y. Zhang, N. Beheshti, L. Beliveau, G. Lefebvret, R. Manghirmalani, R. Mishra, R. Patney, M. Shirazipour, R. Subrahmaniam, C. Truchan, and M. Tatipamula, "StEERING: A software-defined networking for inline service chaining," in *International Conference on Network Protocols*, 2013.

[4] X. Jin, L. E. Li, L. Vanbever, and J. Rexford, "SoftCell: Scalable and flexible cellular core network architecture," in *ACM SIGCOMM CoNext Conference*, Dec. 2013.

[5] 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 *USENIX NSDI*, Apr. 2014.

[6] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, V. Sekar, and A. Akella, "Stratos: A network-aware orchestration layer for virtual middleboxes in clouds." `http://arxiv.org/abs/1305.0209`, 2014.

[7] B. Anwer, T. Benson, N. Feamster, and D. Levin, "Programming Slick network functions," in *Symposium on SDN Research*, June 2015.

[8] Z. A. Qazi, P. Krishna, V. Sekar, V. Gopalakrishnan, K. Joshi, and S. Das, "KLEIN: A minimally disruptive design for an elastic cellular core," in *Symposium on SDN Research*, Mar. 2016.

[9] 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 *ACM SIGCOMM*, pp. 13–24, 2012.

[10] "Faster DDoS mitigation with increased customer control: Introducing Verisign OpenHybrid customer activated mitigation," Sept. 2015. `http://blogs.verisign.com/blog/entry/faster_ddos_mitigation_with_increased`.

[11] C. Raiciu, C. Paasch, S. Barre, , A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley, "How hard can it be? designing and implementing a deployable Multipath TCP," in *Networked Systems Design and Implementation*, 2012.

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

[13] W. Haeffner, J. Napper, M. Stiemerling, D. Lopez, and J. Uttaro, "Service Function Chaining Use Cases in Mobile Networks." IETF Internet Draft draft-ietf-sfc-use-case-mobility-07, Oct. 2016.

[14] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "OpenNF: Enabling innovation in network function control," in *ACM SIGCOMM*, pp. 163–174, 2014.

[15] S. Rajagopalan, D. Williams, H. Jamjoon, and A. Warfield, "Split/Merge: System support for elastic execution in virtual middleboxes," in *USENIX NSDI*, Apr. 2013.

[16] "IETF Working Group on Service Function Chaining (SFC)." `http://datatracker.ietf.org/wg/sfc/`.

[17] "Contrail Feature Guide, Release 2.20, Service Chaining." `http://www.juniper.net/techpubs/en_US/contrail2.2/topics/task/configuration/service-chaining-vnc.html`.

[18] "Neutron service insertion and chaining." `http://wiki.openstack.org/wiki/Neutron/ServiceInsertionAndChaining`.

[19] A. C. Snoeren and H. Balakrishnan, "An end-to-end approach to host mobility," in *ACM MOBICOM*, 2000.

[20] E. Nordström, D. Shue, P. Gopalan, R. Kiefer, M. Arye, S. Ko, J. Rexford, and M. J. Freedman, "Serval: An end-host stack for service-centric networking," in *USENIX NSDI*, April 2012.

[21] 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 Communications Surveys and Tutorials*, vol. 12, pp. 186–204, April 2010.

[22] R. Atkinson, S. Bhatti, and S. Hailes, "Evolving the Internet architecture through naming," *IEEE Journal on Selected Areas in Communication*, vol. 28, pp. 1319–1325, October 2010.

[23] A. R. Natal, L. Jakab, M. Portolés, V. Ermagan, P. Natarajan, F. Maino, D. Meyer, and A. C. Aparicio, "LISP-MN: Mobile networking through LISP," *Wireless Personal Communications*, vol. 70, pp. 253–266, May 2013.

[24] C. Perkins, D. Johnson, and J. Arkko, "Mobility support in IPv6." IETF Request for Comments

6275, July 2011.
[25] P. Zave and J. Rexford, "The design space of network mobility," in *Recent Advances in Networking* (O. Bonaventure and H. Haddadi, eds.), ACM SIGCOMM, 2013.
[26] C. Paasch and O. Bonaventure, "Multipath TCP: Decoupled from IP, TCP is at last able to support multihomed hosts," *ACM Queue*, vol. 12, March 2014.
[27] J. R. Iyengar, P. D. Amer, and R. Stewart, "Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths," *IEEE/ACM Transactions on Networking*, vol. 14, no. 5, pp. 951–964, 2006.
[28] "Docker." `https://www.docker.com/`.
[29] "Mininet: An Instant Virtual Network on your Laptop (or other PC)." `http://mininet.org/`.
[30] G. J. Holzmann, *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.
[31] "Dysco supplemental material." `https://github.com/dysco/`.
[32] "The Bro Network Security Monitor." `https://www.bro.org/`.
[33] "Snort." `https://www.snort.org/`.
[34] "PRADS." `http://gamelinux.github.io/prads/`.
[35] "Linux TC." `http://lartc.org/manpages/tc.txt`.
[36] "Linux netfilter." `http://www.netfilter.org`.
[37] "Suricata." `http://www.suricata-ids.org/`.
[38] "HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer." `http://www.haproxy.org/`.
[39] "Squid." `http://www.squid-cache.org/Intro/`.
[40] "NGINX: A High-Performance HTTP Server and Reverse Proxy." `https://nginx.com/`.
[41] "wrk: A HTTP Benchmarking Tool." `https://github.com/wg/wrk/`.
[42] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/Merge: System support for elastic execution in virtual middleboxes," in *USENIX Symposium on Networked Systems Design and Implementation*, pp. 227–240, 2013.
[43] "Arbor Networks SP Solution," 2015. `http://www.arbornetworks.com/images/documents/Data%20Sheets/DS_SP_EN.pdf`.
[44] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker, "Middleboxes no longer considered harmful," in *OSDI*, pp. 15–15, 2004.
[45] S. Guha and P. Francis, "An end-middle-end approach to connection establishment," in *ACM SIGCOMM*, pp. 193–204, August 2007.
[46] D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, D. Lopez, K. Papagiannaki, P. R. Rodriguez, and P. Steenkiste, "Multi-context TLS (mcTLS): Enabling secure in-network functionality in TLS," in *ACM SIGCOMM*, 2015.
[47] C. Nicutar, C. Paasch, M. Bagnulo, and C. Raiciu, "Evolving the Internet with connection acrobatics," in *Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, ACM, 2013.
[48] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: A framework for NFV applications," in *Symposium on Operating Systems Principles*, ACM, 2015.