Dynamic Service Chaining with Dysco

Pamela Zave
AT&T Labs–Research
pamela@research.att.com

Ronaldo A. Ferreira
UFMS
raf@facom.ufms.br

Xuan Kelvin Zou
Google
kelvinzou@google.com

Masaharu Morimoto
NEC Corporation of America
m-morimoto@bc.jp.nec.com

Jennifer Rexford
Princeton University
jrex@cs.princeton.edu

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 Linux-based prototype, we show that Dysco is provably correct, highly scalable, and able to reconfigure service chains across a range of middleboxes.

CCS CONCEPTS

• **Networks → Network protocols**; **Middleboxes / network appliances**; *Session protocols*; *Network components*;

KEYWORDS

Session Protocol; NFV; Verification; Spin.

ACM Reference format:
Pamela Zave, Ronaldo A. Ferreira, Xuan Kelvin Zou, Masaharu Morimoto, and Jennifer Rexford. 2017. Dynamic Service Chaining with Dysco. In *Proceedings of the ACM SIGCOMM 2017 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, Los Angeles, CA, USA, August 21–25, 2017 (SIGCOMM '17),* 14 pages.
https://doi.org/10.1145/3098822.3098827

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 warrants a re-examination of approaches, as has happened with mobility [49].

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 [1, 9, 10, 18, 19, 36, 37, 50]. 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 [40] or other third-party providers [45], 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 [25].
- A multihomed host spreads traffic over multiple administrative domains (e.g., enterprise WiFi and commercial cellular network),

SIGCOMM '17, August 21–25, 2017, Los Angeles, CA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4653-5/17/08…$15.00
https://doi.org/10.1145/3098822.3098827

yet some middleboxes need to see all the data in a TCP session (e.g., for parental controls [38]). 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 [10, 18, 37], maintain session affinity [10, 18], identify packets whose headers have been changed by a middlebox [9, 10, 37], install forwarding rules for modified packets [10], and allow classification by middleboxes [9]. 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 [6, 16, 26]. 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 [3, 4, 24, 29, 30, 34, 41] and multihoming [17, 31], 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 [20]):

- 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 [33].
- 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, ongoing video sessions could be redirected through compression middleboxes [13].
- A middlebox that is overloaded or undergoing maintenance, could be replaced with another of the same type (e.g., see [11, 39]).
- 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 relability, 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 the 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 [15],

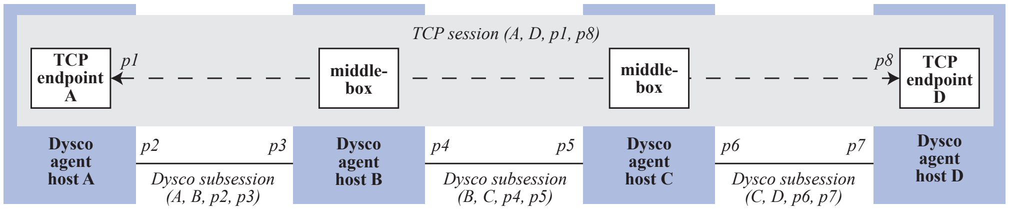


Figure 1: A TCP session with its Dysco subsessions.

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 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 [7]) and experimentation with Mininet [23]. Experiments show that session setup is fast, steady-state throughput is high, and disruption during reconfiguration is small.

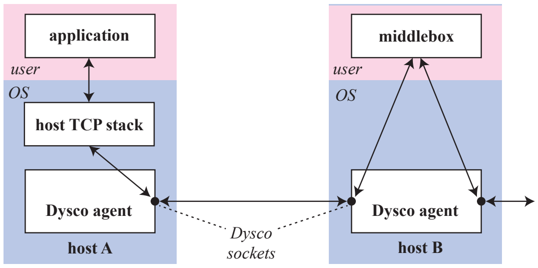


Figure 2: Data flow inside hosts with Dysco agents.

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 section, 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 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. 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 such, Dysco works with existing application-layer protocols. 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.

Establishment of the service chain: Establishment of the service chain in Figure 1 begins when 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 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, 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 [9] and Stratos [10], 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.

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.

2.2 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, including resolving any contention if multiple portions of a service chain try to change at the same time.

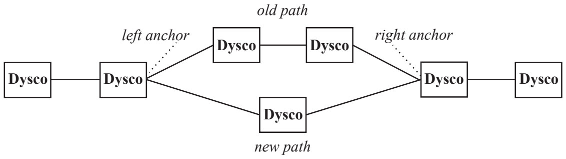


Figure 3: Agents reconfigure a segment of a session, replacing an old path with two middleboxes by a new path with one.

2.3 Agents can reconfigure a session

Reconfiguration of the service chain of a session 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 by exchanging control packets 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.

Security: Like other session protocols [4, 24, 29, 30, 34, 41], Dysco is vulnerable to adversaries that inject or modify control packets. 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 packets) and on-path attacks (e.g., encrypting control packets 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 (see below). 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.

2.4 Sessions and service chains need not coincide exactly

In Figure 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.

One option is that a service chain can 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 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 (e.g., a "splice" call to relegate further handling of the traffic to the kernel). 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.

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 packet 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 §3.4). 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 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.

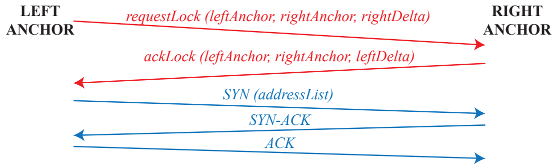

LEFT ANCHOR
RIGHT ANCHOR
requestLock (leftAnchor, rightAnchor, rightDelta)
ackLock (leftAnchor, rightAnchor, leftDelta)
SYN (addressList)
SYN-ACK
ACK

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

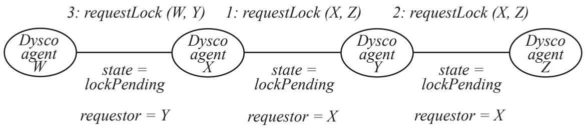


Figure 5: Contention to reconfigure overlapping segments.

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.

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.

3.3 Control signaling

Dynamic reconfiguration requires control signaling, e.g., to resolve contention over segments (§3.2) and to cancel reconfiguration if a new path cannot be created (§3.6).

In Dysco we implement control signals as UDP packets, rather than introducing extra data into the TCP byte stream. This simplifies implementation, as the Dysco agents do not have to monitor the data stream for control signals, extract them, and fix the TCP header information (such as sequence number and checksum). Also, an implementation in the data path would introduce additional delays to data packets for processing the control information.

Although the packets used to set up the new path (as shown in Figure 4) and tear down the old path resemble TCP SYN and FIN handshakes, they are actually UDP packets. The reason we do not use a TCP SYN handshake (as used in Multipath TCP to set up a new subflow [38]) is as follows. A principal design goal for dynamic reconfiguration is to disrupt data transfer as little as possible. While we attempt to set up the new path, data transfer continues on the old path. This means that the SYN to set up the new path has no initial sequence number, as the cutoff number is not determined until the path is ready to use. This is not a problem for middlebox applications because, not surprisingly, the only middleboxes inserted in a new path are ones that do not need to see the initial SYN handshake. Examples are DPIs that operate at the packet level, and middleboxes in the old path that are being replaced (with migration of the session state, as detailed in §3.5).

The reason that we do not use an actual FIN handshake to tear down the old path is that, if TCP packets are used, 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 §3.7). In both cases sequence numbers in the packets mean the same thing, so for both cases TCP packets are passed to the application.

3.4 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 with 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*.

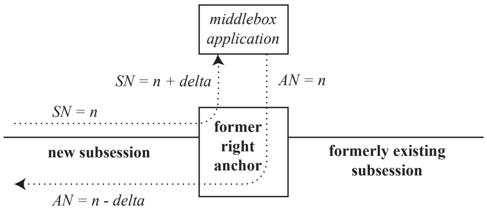


Figure 6: How a former anchor modifies TCP packets (dotted arrows) of the new subsession.

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 below 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

Figure 6 illustrates the use of this table. In the figure, a former right anchor is holding a *delta* (assumed positive in this example) from the reconfiguration, which means that a middlebox formerly to the left of it added *delta* bytes to the data stream. As shown in the figure, packets going into the Dysco agent from the new subsession have *delta* added to their sequence numbers so that the agent's middlebox and all hosts to the right of the agent see consistent sequence numbers. Packets going out of the Dysco agent on the new subsession have *delta* subtracted from their acknowledgment numbers, so that all hosts to the left of the agent see consistent acknowledgment numbers.

3.5 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.

[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.

predicate on *byteSeq*	where to send byte
$\mathit{byteSeq} < \mathit{oldSent}$	old path
$\mathit{byteSeq} \geq \mathit{oldSent}$	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
$\mathit{packetAck} \leq \mathit{oldRcvd} \land$ $\mathit{packetAck} > \mathit{oldRcvdAcked}$	old path
$\mathit{packetAck} > \mathit{oldRcvd} \land$ $\mathit{oldRcvd} = \mathit{oldRcvdAcked}$	new path
$\mathit{packetAck} > \mathit{oldRcvd} \land$ $\mathit{oldRcvd} > \mathit{oldRcvdAcked}$	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 $\mathit{oldSentAcked} = \mathit{oldSent}$. 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:

$\mathit{oldRcvdAcked} = \mathit{oldRcvd} \land \mathit{oldRcvd} = \mathit{firstNewRcvd}$

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 UDP FIN packets on the old path, then clean up the extra state variables.

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 [39]. 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.

3.6 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.

Unfortunately, dynamic reconfiguration cannot be used to recover from the failure of a middlebox. This is because the old path must be fully operational for the protocol to work. Consequently, the utility of reconfiguration is limited to policy change and resource management, rather than fault-tolerance.

3.7 Design and verification

In designing the 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 (§3.1), how to exercise distributed control among conflicting reconfiguration attempts (§3.2), how to compute and use deltas to accommodate the broadest range of middlebox applications (§3.4), how to split acknowledgments across the two paths and determine when the use of the old path is completed (§3.5), and how to handle failures (§3.6). We had to decide whether any particular packet should be TCP or UDP (§3.3). 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 [15]. 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.
- All sessions terminate cleanly.

The model, along with extensive documentation of design, modeling abstractions, and Spin runs, can be found at [8]. It shows that

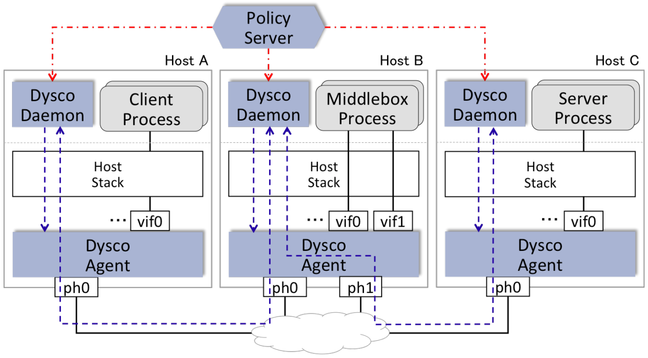


Figure 7: Implementation, where solid black lines represent the data path, blue dashed lines the control path, and red dashed-dotted lines the management path for distributing policies.

with modern tools, protocol design can be more ambitious without sacrificing robust operation.

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 7.

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. The agent could be implemented in various ways, including a modified device driver, a software switch, or a user-space library like DPDK with direct NIC access. 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 [5], Snort [42][2], and PRADS [35]), and middleboxes that run in the Linux kernel (e.g., Traffic Controller (tc) [22] and Iptables/Netfilter Firewall [21]). 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. The kernel module consists 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 [27], HAProxy [14], Iptables/Netfilter [21], Linux tc [22], and libpcap-based middleboxes. Most middleboxes send and receive data via `libpcap`, user socket, or Linux `sk_buff`. Some middleboxes only read the packets (e.g., PRADS [35], Bro [5], Snort [42], Suricata [44], Linux tc [22], Iptables/Netfilter Firewall [21]) while some others modify the TCP session identifier or sequence numbers (e.g., Iptables/Netfilter NAT [21], HAProxy [14], Squid [43]). Middleboxes that only read the packets and 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 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 [11].

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 [8].

4.2 Protocol details

Tagging SYN packets: The local tags added to SYN packets, as described in §2.1, 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

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

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)
```

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. Second, we support unmodified middleboxes that use the Linux's "splice" system call. This system call is used by many applications, *e.g.,* HAProxy [14], to avoid transferring data from the kernel to user space. For this case, we provide a shared library that intercepts 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 associated TCP sessions (i.e., the two five-tuples). 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. Note that we assume that the application calling splice does not want to process the data anymore. This is the case for L7 load balancers, but this is not necessarily true when an HTTP proxy is handling persistent connections, so the shared library must be used with prudence.

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.

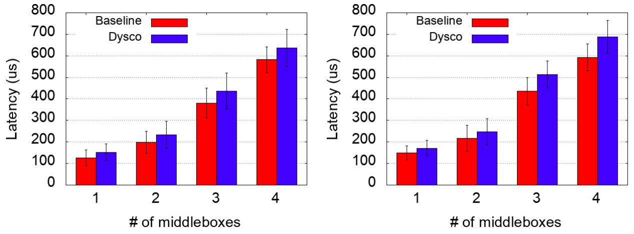
5 PERFORMANCE EVALUATION

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 SYN packets. 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.

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 layer-two switches, forming two independent LANs. The 11 hosts run Ubuntu Linux with kernel 4.4.0.

5.1 Session initiation

Figure 8 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 8(a) shows the latencies when the checksum computation is offloaded to the NIC and Figure 8(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 94μs. 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 8: Latency for session initiation.

5.2 Data-plane throughput

Figure 9 shows the goodput, measured at the receivers, of multiple TCP sessions between four clients and four servers connected via a middlebox that simply 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 10000 sessions. Note that after 100 sessions the link becomes the bottleneck, so we do not notice a significant difference between Dysco and the baseline. Receive side scaling (RSS) is supported in the NIC and enabled in the Linux kernel, so packets belonging to one TCP session are directed to the same core. Therefore, the result for one session gives a better idea of the performance degradation of Dysco, because it represents the goodput of one core.

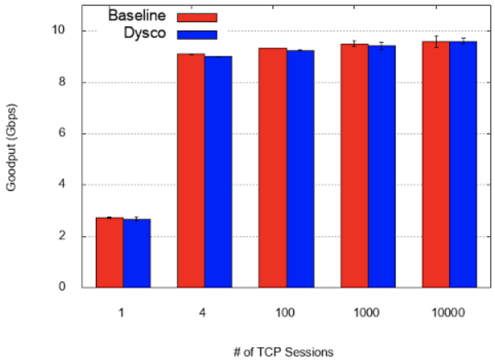


Figure 9: Goodput of Dysco compared with the baseline.

We also measured the number of requests that NGINX [27], a popular HTTP server, can sustain under Dysco and compared the results with the baseline. The measurement was performed with `wrk` [47], an HTTP benchmarking tool, with 16 threads and four hundred persistent connections, as recommended in [47]. 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.

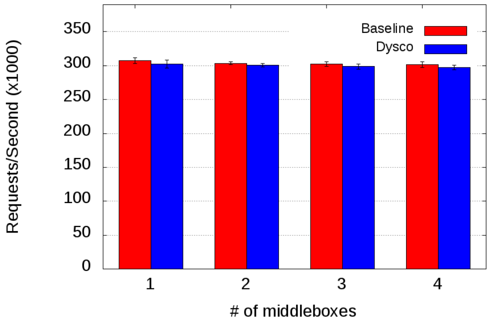


Figure 10: Number of HTTP requests per second NGINX can serve under Dysco and the baseline.

5.3 Dynamic reconfiguration

In this section, we investigate a few scenarios of dynamic reconfiguration. We use the logical topology of Figure 11; one of the hosts works as the router, and each IP subnet is on a different VLAN.

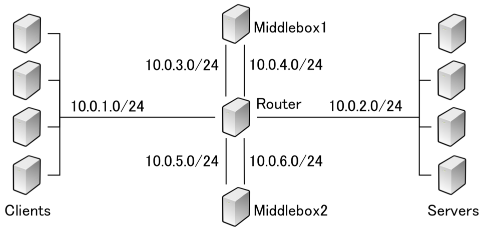


Figure 11: Testbed topology for the performance evaluation of the reconfiguration experiments.

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 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 150 TCP sessions for a total of 600 simultaneous sessions.

The top of Figure 12 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

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 12 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.

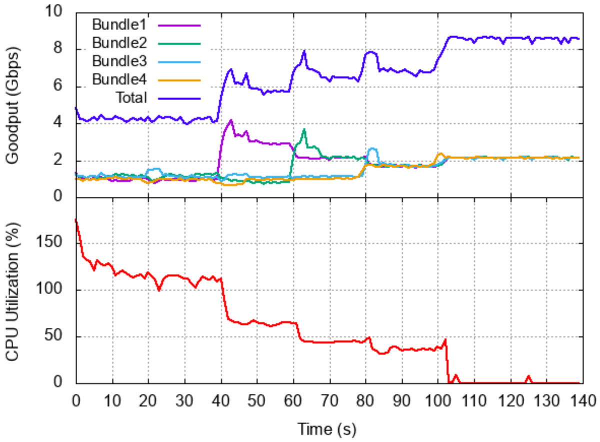


Figure 12: Goodput of TCP sessions (top) and CPU Utilization at the proxy (bottom) before and after multiple reconfigurations.

We can see in Figure 12 that the reconfigurations are successful and the traffic reaches steady-state behavior after 100 seconds. During the reconfiguration, the Dysco agent on the proxy advertises a small window to the senders to reduce the amount of traffic on the receivers. Note that during the 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 strategy that worked best was to advertise the minimum of the actual advertised window and a small constant (64K) that allowed the flow of packets to continue without overwhelming the receivers. Figure 13 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.

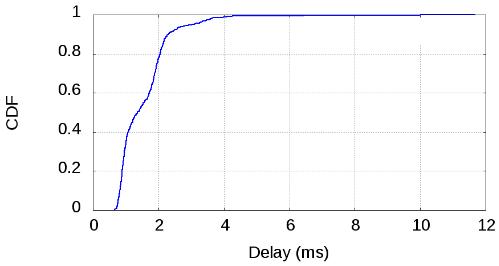
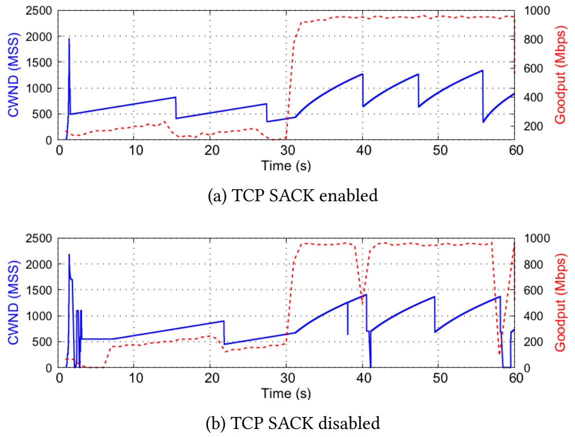


Figure 13: CDF of the reconfiguration time for the proxy removal.

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 14(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 14(b) shows why the Dysco agents must handle TCP options—with TCP SACK disabled, packet losses temporarily degrade session performance.



(a) TCP SACK enabled

(b) TCP SACK disabled

Figure 14: TCP performance during reconfiguration.

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 [11, 39]. 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 11. 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.

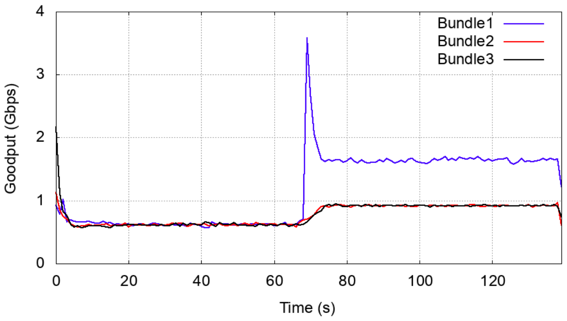


Figure 15: Goodput during reconfiguration and state migration.

Figure 15 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 11. The time series represents the goodput measured at one-second intervals. The blue 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 blue-line sessions are 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 sessions that migrate from Middlebox1 to Middlebox2 do not suffer performance degradation (i.e., no lost or reordered packets) and are 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 13, we can see that in this case the state transfer dominates the reconfiguration time.

6 LIMITATIONS

Our prototype does not implement the security mechanisms presented in §2.3.

In principle, Dysco service chains can span domain boundaries, because packets are steered from one middlebox to another by ordinary addressing and forwarding. In practice, a SYN packet with a payload—the address list, as in §2.1—may not be accepted by a domain's firewall.

There is also a potential practical problem with service chains that begin where a TCP session enters a new domain. Routing in the domain must ensure that all packets of the session are routed to the same middlebox, one with a Dysco agent that initiates the service chain for the session.

There may be an additional difficulty with dynamic reconfiguration, if the reconfigured segment crosses a domain boundary, and an unmodified NAT at the boundary does not allow UDP packets to pass through. Our current prototype has yet another problem that the new path is set up with UDP packets (§3.3), so its TCP data packets appear unsolicited. The latter problem can be fixed, however, by setting up the new path with a TCP SYN handshake with random initial sequence numbers, and using the delta mechanism (§3.4) to correct subsequent sequence numbers in the data stream.

Most of these difficulties have a single root cause: service chaining in Dysco, and especially dynamic reconfiguration, requires control signaling and metadata that are difficult to transmit within a TCP session. This is exactly the same difficulty encountered in implementing multihoming with Multipath TCP [38]. Multipath TCP uses primarily TCP options, which we did not use for lack of space. Whether a protocol uses SYN payloads, TCP options, or auxiliary UDP signaling, there is a danger that control and metadata will be blocked, dropped, or modified in transit. This can happen because of security measures, or, in the case of TCP options, because of innocent functions such as resegmentation in NICs. The safest approach seems to be encoding control and metadata in escape sequences in the TCP byte stream, but this is inefficient and requires significant manipulation of sequence numbers.

Given the importance of TCP, this is a problem urgently in need of a good solution. There should be a secure, reliable, and efficient way of associating control and metadata with a TCP session. Most importantly, designers of this solution should recognize that there may be more than one additional feature, function, or protocol adding metadata to a single session, so that they do not interfere with each other.

7 RELATED WORK

7.1 Service chaining by forwarding

BGP: Early solutions to dynamic service chaining manipulate BGP to "hijack" traffic, either within a single domain [2] or across the wide area [45]. 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.

Stratos and E2: Stratos [10] and E2 [32] are designed for middlebox deployment within clouds. They use fine-grained forwarding rules for (static) service chaining, inheriting the scaling challenges mentioned in §1. 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 middlebox management, including these.

OpenNF: OpenNF [11] (and also Split-Merge [39]) 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.

7.2 Service chaining by session protocols

DOA: Like Dysco, DOA [46] 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 middleboxes inserted on behalf of 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 [12], 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. [28] 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 [16] 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.

7.3 Research complementary to Dysco

mcTLS: Multi-context TLS (mcTLS) [25] 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 the relevant security information for decrypting and reencrypting the data. Like Dysco, mcTLS has a list of middleboxes in a session setup message, which is yet another example of the need for metadata. 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 [49]. Of these, ECCP [3], TCP Migrate [41], and msocket [48] are TCP-oriented. Both ECCP and TCP Migrate are oblivious to middleboxes. msocket explicitly uses signaling at the application layer to avoid introducing new TCP options and to deal with the complexities introduced by middleboxes. Application data are encapsulated in msocket packets, so data streams look like regular TCP data for middleboxes. Likewise signaling protocols have been used for supporting multihoming, notably ECCP [3] and Multipath TCP [31]. All of these protocols are intrinsically compatible with Dysco, which suggests that merging the approaches would be fruitful.

8 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 (§1). 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 as a mechanism for service chaining (which has little hope of extending to inter-domain service chains). 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?"

ACKNOWLEDGMENTS

We thank our shepherd Vyas Sekar and the anonymous SIGCOMM reviewers for their valuable feedback. We also thank Mina Arashloo, Bharath Balasubramanian, Jennifer Gossels, Rob Harrison, Yaron Koral, Robert MacDavid, and Shankaranarayanan Narayanan for their feedback on earlier drafts of this paper. This work was supported in part by NSF grant CNS-116112, and by the Brazilian National Council for Scientific and Technological Development (CNPq) proc. 201983/2014-1.

REFERENCES

[1] Bilal Anwer, Theophilus Benson, Nick Feamster, and David Levin. 2015. "Programming Slick Network Functions". In *ACM SIGCOMM Symposium on Software Defined Networking Research.* 14:1–14:13.

[2] Arbor 2015. Arbor Networks SP Solution. (2015). http://www.arbornetworks.com/images/documents/Data%20Sheets/DS_SP_EN.pdf.

[3] Matvey Arye, Erik Nordstrom, Robert Kiefer, Jennifer Rexford, and Michael J. Freedman. 2012. A Formally-verified Migration Protocol for Mobile, Multi-homed Hosts. In *IEEE International Conference on Network Protocols.* 1–12.

[4] R. Atkinson, S. Bhatti, and S. Hailes. 2010. Evolving the Internet Architecture Through Naming. *IEEE Journal on Selected Areas in Communication* 28, 8 (October 2010), 1319–1325.

[5] Bro 2017. The Bro Network Security Monitor. (2017). https://www.bro.org/.

[6] Contrail 2017. Contrail Feature Guide, Release 2.20, Service Chaining. (2017). http://www.juniper.net/techpubs/en_US/contrail2.2/topics/task/configuration/service-chaining-vnc.html.

[7] Docker 2017. Docker. (2017). https://www.docker.com/.

[8] Dysco 2017. Dysco supplemental material. (2017). https://github.com/dysco/.

[9] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. 2014. Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags. In *USENIX Conference on Networked Systems Design and Implementation*. 533–546.

[10] Aaron Gember, Anand Krishnamurthy, Saul St. John, Robert Grandl, Xiaoyang Gao, Ashok Anand, Theophilus Benson, Vyas Sekar, and Aditya Akella. 2014. Stratos: A Network-Aware Orchestration Layer for Virtual Middleboxes in Clouds. http://arxiv.org/abs/1305.0209. (2014).

[11] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. 2014. OpenNF: Enabling Innovation in Network Function Control. In *ACM SIGCOMM Conference on Applications, Technologies, and Protocols for Computer Communication*. 163–174.

[12] Saikat Guha and Paul Francis. 2007. An End-Middle-End Approach to Connection Establishment. In *ACM SIGCOMM Conference on Applications, Technologies, and Protocols for Computer Communication*. 193–204.

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

[14] HAProxy 2017. HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer. (2017). http://www.haproxy.org/.

[15] Gerard J. Holzmann. 2003. *The Spin Model Checker: Primer and Reference Manual* (first ed.). Addison-Wesley Professional, Boston, MA, USA.

[16] IETF 2017. IETF Working Group on Service Function Chaining (SFC). (2017). http://datatracker.ietf.org/wg/sfc/.

[17] J. R. Iyengar, P. D. Amer, and R. Stewart. 2006. Concurrent Multipath Transfer Using SCTP Multihoming over Independent End-to-End Paths. *IEEE/ACM Transactions on Networking* 14, 5 (2006), 951–964.

[18] Xin Jin, Li Erran Li, Laurent Vanbever, and Jennifer Rexford. 2013. SoftCell: Scalable and Flexible Cellular Core Network Architecture. In *ACM Conference on Emerging Networking Experiments and Technologies*. 163–174.

[19] Dilip A. Joseph, Arsalan Tavakoli, and Ion Stoica. 2008. A Policy-Aware Switching Layer for Data Centers. In *ACM SIGCOMM Conference on Applications, Technologies, and Protocols for Computer Communication*. 51–62.

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

[21] Linux-NF 2017. Linux Netfilter. (2017). http://www.netfilter.org.

[22] Linux-TC 2017. Linux TC. (2017). http://lartc.org/manpages/tc.txt.

[23] Mininet 2017. Mininet: An Instant Virtual Network on your Laptop (or other PC). (2017). http://mininet.org/.

[24] A. R. Natal, L. Jakab, M. Portolés, V. Ermagan, P. Natarajan, F. Maino, D. Meyer, and A. C. Aparicio. 2013. LISP-MN: Mobile networking through LISP. *Wireless Personal Communications* 70, 1 (May 2013), 253–266.

[25] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego Lopez, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. 2015. Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *ACM SIGCOMM Conference on Applications, Technologies, and Protocols for Computer Communication*. 199–212.

[26] Neutron 2017. Neutron Service Insertion and Chaining. (2017). http://wiki.openstack.org/wiki/Neutron/ServiceInsertionAndChaining.

[27] NGINX 2017. NGINX: A High-Performance HTTP Server and Reverse Proxy. (2017). https://nginx.com/.

[28] Catalin Nicutar, Christoph Paasch, Marcel Bagnulo, and Costin Raiciu. 2013. Evolving the Internet with Connection Acrobatics. In *Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. 7–12.

[29] P. Nikander, A. Gurtov, and T. R. Henderson. 2010. Host Identity Protocol (HIP): Connectivity, Mobility, Multi-Homing, Security, and Privacy over IPv4 and IPv6 Networks. *IEEE Communications Surveys and Tutorials* 12, 2 (April 2010), 186–204.

[30] Erik Nordström, David Shue, Prem Gopalan, Rob Kiefer, Matvey Arye, Steven Ko, Jennifer Rexford, and Michael J. Freedman. 2012. Serval: An End-host Stack for Service-centric Networking. In *USENIX Conference on Networked Systems Design and Implementation*. 85–98.

[31] Christoph Paasch and Olivier Bonaventure. 2014. Multipath TCP. *Commun. ACM* 57, 4 (April 2014), 51–57.

[32] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. E2: A Framework for NFV Applications. In *Symposium on Operating Systems Principles*. 121–136.

[33] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. 2013. Ananta: Cloud Scale Load Balancing. In *ACM SIGCOMM Conference on Applications, Technologies, and Protocols for Computer Communication*. 207–218.

[34] C. Perkins, D. Johnson, and J. Arkko. 2011. Mobility Support in IPv6. IETF Request for Comments 6275. (July 2011).

[35] PRADS 2017. PRADS. (2017). http://gamelinux.github.io/prads/.

[36] Zafar Ayyub Qazi, Phani Krishna, Vyas Sekar, Vijay Gopalakrishnan, Kaustubh Joshi, and Samir Das. 2016. KLEIN: A Minimally Disruptive Design for an Elastic Cellular Core. In *ACM Symposium on SDN Research*. 2:1–2:12.

[37] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. 2013. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *ACM SIGCOMM Conference on Applications, Technologies, and Protocols for Computer Communication*. 27–38.

[38] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. 2012. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *USENIX Conference on Networked Systems Design and Implementation*. 399–412.

[39] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. 2013. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *USENIX Conference on Networked Systems Design and Implementation*. 227–240.

[40] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. 2012. Making Middleboxes Someone Else’s Problem: Network Processing As a Cloud Service. In *ACM SIGCOMM Conference on Applications, Technologies, and Protocols for Computer Communication*. 13–24.

[41] Alex C. Snoeren and Hari Balakrishnan. 2000. An End-to-End Approach to Host Mobility. In *ACM Annual International Conference on Mobile Computing and Networking*. 155–166.

[42] Snort 2017. Snort. (2017). https://www.snort.org/.

[43] Squid 2017. Squid. (2017). http://www.squid-cache.org/Intro/.

[44] Suricata 2017. Suricata. (2017). http://www.suricata-ids.org/.

[45] Verisign 2015. 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.

[46] Michael Walfish, Jeremy Stribling, Maxwell Krohn, Hari Balakrishnan, Robert Morris, and Scott Shenker. 2004. Middleboxes No Longer Considered Harmful. In *USENIX Symposium on Opearting Systems Design & Implementation*. 215–230.

[47] wrk 2017. wrk: A HTTP Benchmarking Tool. (2017). https://github.com/wg/wrk/.

[48] Aditya Yadav and Arun Venkataramani. 2016. msocket: System Support for Mobile, Multipath, and Middlebox-Agnostic Applications. In *IEEE International Conference on Network Protocols*. 1–10.

[49] Pamela Zave and Jennifer Rexford. 2013. The Design Space of Network Mobility. In *Recent Advances in Networking*, Olivier Bonaventure and Hamed Haddadi (Eds.). ACM SIGCOMM, New York, NY, USA.

[50] Ying Zhang, Neda Beheshti, Ludovic Beliveau, Geoffrey Lefebvret, Ravi Manghirmalani, Ramesh Mishra, Ritun Patney, Meral Shirazipour, Ramesh Subrahmaniam, Catherine Truchan, and Mallik Tatipamula. 2013. StEERING: A Software-Defined Networking for Inline Service Chaining. In *IEEE International Conference on Network Protocols*. 1–10.