Don't Call Them Middleboxes, Call Them Middlepipes

Hani Jamjoom Dan Williams Upendra Sharma
IBM T. J. Watson Research Center
Yorktown Heights, NY

Abstract

As interest grows in introducing network functions—middleboxes—to Platform as a Service (PaaS) clouds, it is tempting to treat them as normal PaaS services. However, the PaaS service abstraction lacks sufficient support for middlebox services. For example, network functions may require proximity to data sources for efficient snooping or request rewriting, or access to raw packets rather than application-level requests. Instead, we propose a new network function abstraction to PaaS clouds called *middlepipes.* True to PaaS philosophy, middlepipes are sufficiently high level for application developers to forget about details like packets vs. requests and data source proximity. Middlepipes can be chained together to cooperatively interpose on traffic between applications and services. Furthermore, they can generate callbacks into applications; in this paper, we describe the middlepipe PaaS architecture in the context of a "circuit breaker" network function.

Categories and Subject Descriptors

C.2.1 [**COMPUTER-COMMUNICATION NETWORKS**]: Network Architecture and Design

Keywords

Middlebox; Network Function Virtualization; Platform as a Service; Cloud Computing

1. INTRODUCTION

Unlike Infrastructure as a Service clouds, PaaS clouds promise their users a higher-level abstraction in which applications express interest in services and runtimes (e.g., "I want a Ruby runtime that talks to a MongoDB service"). In this example, the PaaS would near instantly provision the necessary Ruby runtime containers and MongoDB instances. This, in principle, frees developers from worrying about low-level details such as virtual machine (VM) configurations, networking issues, etc. Even higher level capabilities—like elasticity and high-availability—are implicitly assumed, rather than explicitly managed.

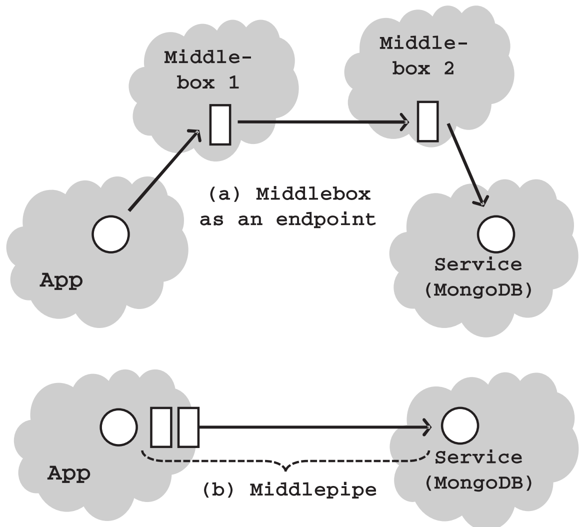

Figure 1: Middlebox as an endpoint vs. Middlepipes

Not surprisingly, today's PaaS vision emphasizes offering all its capabilities in bite-size services: its runtimes, databases, etc. These services are conceptually treated as endpoints that have four simple lifecycle calls: *provision*, *bind*, *unbind*, and *deprovision.* Even network functions (NFs) and middleboxes are expected to be offered in the same way [13, 22]. We argue that this—endpoint—service model is not suitable for all network functions. Sophisticated features (e.g., intrusion detection, response caching, circuit breaking, transcoding, etc.) cannot be easily implemented as endpoint services that interpose on requests and network packets. They would also break the simplicity of consuming PaaS capabilities, where developers would need to configure and manage these middlebox endpoints, worrying about the logical topology of the application and middleboxes. As depicted in Figure 1(a), if a developer wants to add a middlebox (e.g., an intrusion detection system) between a Ruby front-end and a MongoDB, he/she would need to provision the middlebox endpoint; then he/she would need to wire the logical flow between various components. A sophisticated load balancer—that enables A/B testing or incremental upgrade—would require even more configuration complexity by the developer.

Instead, we advocate that NFs and middleboxes represent alternative connectivity options within a PaaS. Conceptually, this implies that NFs and middleboxes should be treated as pipes rather than endpoints (Figure 1(b)). We call these *middlepipes*. Middlepipes enable developers to modify how different services and runtimes talk to each other. A developer can for example say "I want to protect all communication between my Ruby runtime and MongoDB using Bro." The PaaS sets up, routes, scales, and maintains the necessary connections between the various application components.

We examine Netflix's OSS components [9] as target use cases for middlepipes. The Netflix OSS includes a set of network-centric components—implemented at the application level—that Netflix uses to run its core infrastructure on top of different clouds. Hystrix [5], for example, is a Java framework for providing circuit breaker capability: the ability to switch services when failure is detected. Hystrix is used to limit failure propagation and, at the same time, provide end-users with default behavior (e.g., default movie recommendation) when a service fails. Zuul [11] is another service that offers request throttling, A/B testing, and stress testing. We focus on these services for three reasons: (1) they capture real needs by a cloud-scale applications, (2) they are open source, and (3) they are currently language specific, requiring heavy application modification. Our goal is to offer similar features in the form of middlepipe services. In the case of the Hystrix middlepipe equivalent, a user should be able to ask the PaaS to use the circuit breaker middlepipe, automatically triggering a callback when service failure is detected.

In this paper, we provide a blueprint for creating and managing middlepipes. We have implemented an early prototype of the middlepipe framework inside Cloud Foundry, a leading open source PaaS. Our approach separates each middlepipe into two parts: *filtering* and *aggregation*. Middlepipe filters are lightweight modules that are instantiated inside each PaaS container (e.g., Warden[1] in the case of Cloud Foundry). Filters share the same design principles of `netfilter`, but allow a greater flexibility of interposing on both requests and network packets. Additionally, filters can be strung together and hot swapped depending an application's needs. To maintain implementation efficiency, filters asynchronously communicate with an aggregator service. The aggregator service understands how to control the various instances of the filters and how to combine any monitoring data across them. Using our framework, we show how to implement a circuit breaker middlepipe. We highlight the implementation challenges for maintaining high throughput between application components, correctly handling callbacks, and allowing dynamic chaining of filters.

2. TODAY'S PAAS

"Platform as a service" like "cloud" is an overloaded term. In this paper, by PaaS, we refer to an offering that manages application runtimes (e.g., Ruby, NodeJS, Java, etc.) and provides simplified access to *services* (e.g., MongoDB, RabbitMQ, etc.). Cloud Foundry [2], Heroku [4], IBM BlueMix [6], Microsoft Azure [10], and Google AppEngine [12] are examples of such a PaaS. Figure 2 describes the architecture of CloudFoundry and Heroku PaaS platforms. These platforms adopt a loosely-coupled service model, wherein each service is composed of five elements from the PaaS point of view: a service metadata catalog and four service lifecycle calls, namely (1) provision (2) bind, (3) unbind and (4) deprovision.

[1]Warden can be thought of as an OS-level container

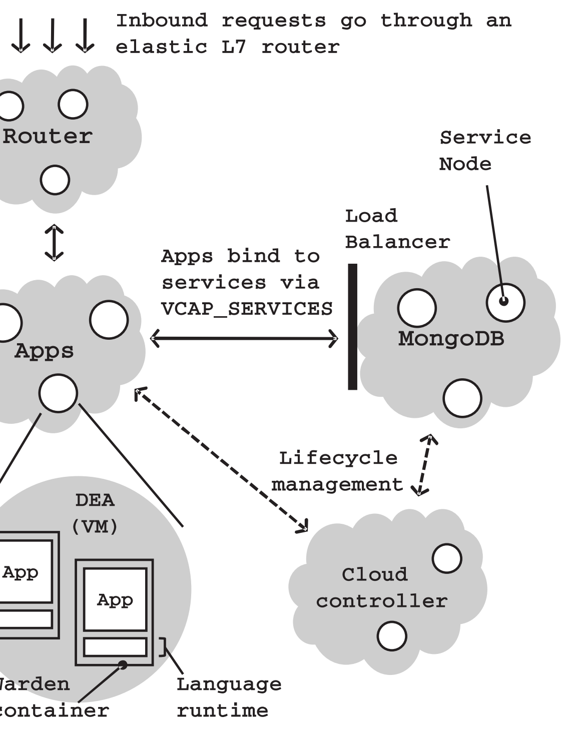

Figure 2: Cloud Foundry PaaS Overview

2.1 An "as a Service" Obsession

In PaaS environments, the service abstraction reduces the complexity for application developers. For example, in Cloud Foundry, users express interest in services and runtimes using `cf`, a simple-to-use PaaS command line interface. A user can request a runtime for his/her application using the command `cf push` and can create a service using the command `cf create-service`. Users can also create bindings between these environments using the command `cf bind-service`. Under the covers, the PaaS manages the number of instances and routing of requests. It should be noted that logical topologies are not typically enforced (i.e., a developer cannot force requests through a service, like a middlebox, unless he/she embeds the logic inside the application).

In current PaaS offerings, network functions are being shoehorned into the service model.[2] In general, this model lacks the necessary support for network functions. The next subsection illustrates these deficiencies through two examples.

[2]For example, in BeanStalk (Amazon's PaaS), developers explicitly request an elastic load balancer and manage it via command line tools, in a similar way to a physical loadbalancer. Additionally, many PaaS providers provide message queuing services, which can be categorized as very high-level network functions.

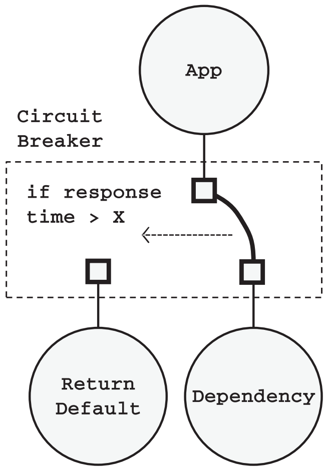

Figure 3: Basic Circuit Breaker Pattern

2.2 Examples

Intrusion Detection System

An intrusion detection system (IDS) monitors the network for suspicious or policy-violating traffic and reports it to a management server. Examples include Bro [17] and Snort [20]. Policies can be specified based on packet-level headers. The IDS examines packets and maintains state about network flows.

We study IDSs inside the PaaS because they are essential middlebox capabilities for many enterprise applications. To implement an IDS in the service model is fundamentally difficult because the service model operates at network requests (e.g., REST requests) instead of network packets. Packet-level headers are hidden from all applications and services.

Circuit Breaker

A key goal for the Netflix OSS is to build highly elastic and highly fault tolerant infrastructure. The circuit breaker design pattern (which is part of Hystrix [5]) provides an interesting approach for fault isolation and better user experience. The idea is to detect failure quickly and isolate the failed component. For example, if a recommendation service fails, Netflix would return a default movie recommendation rather than have the user wait to get customized recommendations.[3]

Figure 3 depicts the high level design of a circuit breaker. A monitoring component listens on requests to various services. When “failure” is detected, the failing component is switched out of the dependency path. Here, failure can include any number of metrics: fail stop, an abnormal response, or a slow response time. After the circuit breaker is tripped, the source is notified for quick failure recovery, possibly returning a cached, default, or empty response. In the case of writes, the system can queue them for future commits.[4]

Circuit breaker requires non-trivial interposition on the invocation path of requests. Specifically, (1) it requires active—stateful—monitoring of requests, (2) when the breaker is tripped, it can trigger a wide spectrum of responses (from a trivial empty response to a more elaborate triggering of exceptions), and (3) it adds a valuable—and currently unavailable—capability to any PaaS.

To implement a circuit breaker in today’s PaaS, particularly the service model, would face a number of challenges. A circuit breaker service would need to be carefully designed with a circuit-breaker-specific API to trigger callbacks to the application in case of tripped circuits. It would face performance challenges while interposing on requests, especially if it was not placed near to the application in the data center. It may be difficult to chain together with other network functions. Or, it may require extensive libraries to be written in every supported programming language to circumvent some of these issues.

2.3 Lack of Support for Network Functions

To summarize, by focusing solely on the service model, PaaS offerings do not provide network functions the support they need to function well. In particular, the PaaS must allow network functions to:

- **Move close to data streams.** Network functions must be able to operate without requiring traffic to be diverted back and forth between services running elsewhere in the data center. Ideally, they must be able to examine or modify traffic with minimal data copying.
- **Operate either at the packet or request level.** Some network functions, such as intrusion detection, require access to a low-level packet stream to perform their task. Others, like circuit breaking, operate at the level of REST API calls. A PaaS must support both granularities.
- **Generate callbacks.** Network functions should not need to implement their own custom APIs or libraries to generate callbacks to applications. Every network function will have similar requirements.
- **Easily chain together.** Chaining of functions and middleboxes is a common and natural process in the network. The system should explicitly provide support for chaining.

In the next section, we describe the *middlepipe*, a new PaaS abstraction in which a PaaS can support network functions.

3. MIDDLEPIPES

Rather than thinking of network functions implemented as middleboxes (endpoint services), we advocate application developers begin to view them as *middlepipes*. A middlepipe is a first-class PaaS abstraction that connects applications and endpoint services. Each middlepipe implements a network function in the PaaS.

Users directly create instances of middlepipes (e.g., `cf create-middlepipe breaker`). Users can bind services and apps together with a middlepipe (e.g., `cf bind-middlepipe breaker myapp mongodb`). This implies that all requests

[3]The Netflix OSS does not technically fall under our definition of PaaS because it is implemented as a language framework rather than an as-a-Service offering. However, it contains a number of interesting (and important) features that other PaaS environments can implement.
[4]Netflix assumes an eventual consistency model.

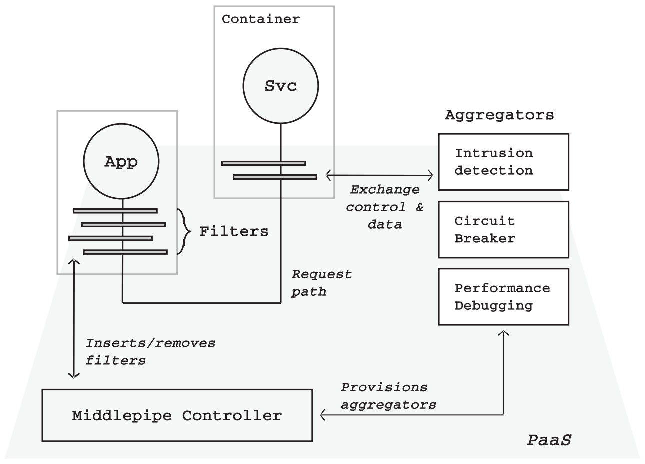

Figure 4: Architecture of Middlepipes

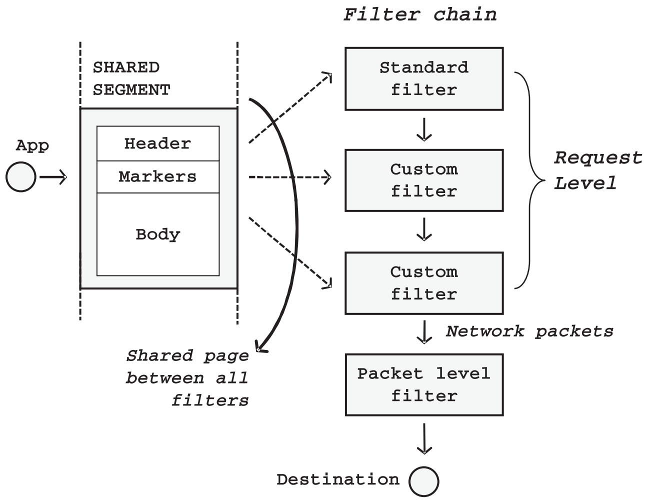

Figure 5: Optimizing Filter Access to Requests

between `myapp` and `mongodb` will go through the `breaker` middlepipe. If another middlepipe provides intrusion detection, then it is bound in a similar way: `cf bind-middlepipe bro myapp mongodb`. Users can update or remove the middlepipes between components or stack different middlepipes atop one another at any point during runtime.

Figure 4 depicts the components inside the PaaS that make up the middlepipe abstraction. Network traffic exiting an application or service flows through a series of *filters*. Filters may implement a network function in its entirety, or may periodically communicate with an *aggregator* to implement the network function. A *scheduler* executes filters on network traffic in an order dictated by a *filter chain*, specified by the user. Finally, a *middlepipe controller* inserts and removes filters, provisions aggregators, and interfaces with a PaaS user.

In the remainder of this section, we describe in more detail how, through the middlepipe abstraction, PaaS offerings can provide support for network functions.

3.1 Moving Close to Data Streams with Filters and Aggregators

Each middlepipe is split into two parts: filters and aggregators. Filters are light-weight modules that are installed in each service instance or runtime container. Filters execute in close proximity to the invocation path. This allows us to (1) distribute computation across the underlying infrastructure, (2) minimize copying of requests and packets, (3) reduce overhead on the network substrate, and (4) simplify billing.

Aggregators are independent services that get provisioned for each user. An aggregator asynchronously interacts with all the filter instances, for example, to collect monitoring information or provide per-middlepipe configuration information. Depending on the middlepipe feature, filters and aggregators can vary in complexity. A dependency monitoring middlepipe, for example, only needs to collect high level statistics about external request invocations. The aggregator can be a monitoring tool like Graphite [3]. A circuit breaker middlepipe, on the other hand, requires that filters are more sophisticated: it must return specific responses when failure is detected. Alternatively, choke points can be implemented with filters and aggregators using synchronous communication.

The middlepipe system does not mandate any particular interface between filters and aggregators. The system, however, automates the creation and destruction of filters as services and containers are scaled up or down, respectively. The system also piggy backs on Cloud Foundry's routing to bind filters to aggregators. Specifically, filters do not need to know where the location or number of instances of the aggregator.

3.2 Operating on Packets and Requests

Components inside a PaaS talk to each other through application-level requests (e.g., REST or API calls). At the same time, network functions and middlepipes can operate purely at the application (request) level, the packet level, or both (e.g., IDS and load balancers inspect packets headers and payloads). We, thus, provide hooks that can interpose on both requests and packets, while minimizing the reconstruction of requests from packets.

Interposing at the packet level is relatively straightforward. As application network traffic enters or exits the application container it passes through the network stack of the underlying OS kernel. In Linux, this presents an opportunity to leverage the `netfilter` packet filtering framework. Through `libnetfilter_queue` [8], user-space applications can examine, modify or instruct the kernel to drop packets.

Relying on `netfilter` is not appropriate for interposition at the request level. Reconstruction of requests from packets (making filters behave like proxies) carries overhead. Instead, we supply language specific libraries (e.g., a modified version of Ruby's `httpclient` gem) and use Cloud Foundry's buildpacks to distribute the libraries. The libraries provide a hook into the request invocation. The library can subsequently transfer the request in its entirety to filters.

There are many possible implementations for the transfer of requests. One efficient approach is for the PaaS to manage shared memory segments containing requests. Figure 5 shows a shared memory segment being used to transfer requests between request-level filters, before the request exits the application stack and enters packet-level filters.

3.3 Callbacks

The circuit breaker middlepipe offers an interesting use case in which the middlepipe can trigger an exception inside the application. This is an example of a callback, which is different from returning a default value, where the application is oblivious to a failure on a downstream component. A middlepipe propagates callback triggers through modification of responses. In particular, middlepipes add *markers* to the response headers in a standard way. Applications check for these markers and perform callbacks based on their value. In the circuit breaker case, the callback will raise an exception.

This facility also creates an opportunity for exceptions to be handled either explicitly during the development of the application or through orthogonal aspects, using any number of aspect oriented programming[5] support packages.

3.4 Chaining

When network traffic enters or exits an application, the middlepipe filter scheduler gains control. The scheduler transfers control over requests or packets to each filter, one at a time. Filters are executed in a sequence according to a chain, similar to `iptables`. Middlepipe chains are user-specified. Our design automatically orders filters such that those operating at the request level are executed before those operating at the packet level. The consolidation of many filters (implementing traditional middlebox functionality) onto a single scheduler is similar to the architecture described in CoMb [21].

4. CIRCUIT BREAKER REVISITED

In this section, we describe how the circuit breaker would be implemented as a middlepipe. For simplicity, we consider the circuit breaker to monitor request latency and "trip the breaker" if latency exceeds a fixed threshold. A request through a tripped breaker results in an exception in the application.

We assume a Ruby application (`App1`) and service (`service1`) both running on Cloud Foundry. An instance of the circuit breaker middlepipe is created via `cf create-middlepipe` and used to connect `App1` and `service1` via `cf bind-middlepipe`. Under the covers, the middlepipe controller instantiates an instance of a circuit breaker filter in the container running `App1` and a circuit breaker aggregation service. The aggregation service maintains a listing of broken circuits that filters periodically query.

`App1` transmits network that are intercepted by the middlepipe scheduler. The middlepipe scheduler executes the circuit breaker filter, which checks its local state to see if the breaker is tripped. If not, the filter saves a timestamp and releases the request back to the scheduler. The scheduler, finding no other filters to run, allows the request to exit the application container. The transfer of control between the application, filters and scheduler are efficient because they are co-located in the same container.

When a response arrives, the middlepipe scheduler once again exerts control and executes the circuit breaker filter. The circuit breaker computes the latency of the request using the previously saved timestamp. Depending on the latency, the filter updates its state and notifies the aggregator. The response is returned to the application.

`App1` receives the response and checks the standard middlepipe headers for any marked return values. In the case that `App1` makes a request and the breaker is tripped, the filter will craft a response with the marked headers communicating that the breaker was tripped.

[5]In aspect oriented programming, code that checks for logged users or performs exception handling can be developed separately from the main body of the application. This, in principle, simplifies adding and maintaining messy code that implement cross cutting concerns separately.

5. RELATED WORK

Our work draws inspiration from a large corpus of research work in industry and academia in the area of cloud computing, middleboxes and NFs. We have closely studied the leading PaaS clouds, namely CloudFoundry [2] and Heroku [4], and have adapted their techniques to simplify the consumption of network functions and middleboxes.

A fair amount of recent work exists on middleboxes [14, 18–20]. None of them are target *aaS environment and, thus, do not address the deployability of middleboxes as a service/solution and/or the ease of doing the same. Service providers like Windows Azure [10], Amazon AWS [1], CloudFoundry, Google App Engine [12], Heroku, offer middlebox services like load balancing across clustered VMs/Apps. They cannot support chaining of various network services. Dixon et al. [16] propose an architecture in which middlebox functionality is moved to the edge of the network and runs on end hosts in an attested environment. Our work takes a similar approach at the PaaS level.

CloudNaaS [15] aims at providing rich set of middlebox functionalities for applications. Each of them require complex fine grain configuration, thereby making its consumption difficult. Our approach, on the other hand, tries to simplify consumption.

APLOMB [22] aims to provide middlebox services to enterprises by outsourcing these functionalities to the cloud. This approach will push the middlebox far from the application, thereby impacting its performance and also the ease of consumption. Our approach derives motivation from the idea of netfilters/ipchains [7] and brings middlebox functionality closer to the application.

6. CONCLUSION

Today's PaaS clouds offer an early glimpse of an environment where the OS and all that is beneath is deemed irrelevant. In such environments, the role of software defined networking and network function virtualization must support the overall design goals: services, runtimes, and bindings. In this paper, we provide our current vision for network functions within a PaaS. Our middlepipe architecture offers developers alternative implementations to the connectivity fabric between services and runtimes. We show how middlepipes can implement valuable services like the circuit breaker design pattern. We are actively developing middlepipe implementations inside Cloud Foundry, and hope that the framework fosters new ways to advance network functionality inside PaaS environments.

7. REFERENCES

[1] Amazon Web Services. `http://aws.amazon.com/`.

[2] Cloud Foundry. `http://docs.cloudfoundry.org`.

[3] Graphite - Scalable Realtime Graphing. http://graphite.wikidot.com/.

[4] Heroku: Cloud Application Platform. https://www.heroku.com/.

[5] Hystrix: Latency and Fault Tolerance for Distributed Systems. https://github.com/Netflix/Hystrix.

[6] IBM Codename: BlueMix. https://ace.ng.bluemix.net/.

[7] Netfilter. http://www.netfilter.org/.

[8] netfilter: libnetfilter-queue. http://www.iptables.org/projects/libnetfilter_queue/index.html.

[9] Netflix Open Source Software. http://netflix.github.io/#repo.

[10] Windows Azure Platform. https://www.windowsazure.com/en-us/.

[11] Zuul. https://github.com/Netflix/zuul/wiki.

[12] Google AppEngine. https://developers.google.com/appengine/, Nov. 2012.

[13] ETSI Network Function Virtualisation (NFV); Use Cases.

[14] Anand, A., Gupta, A., Akella, A., Seshan, S., and Shenker, S. Packet caches on routers: The implications of universal redundant traffic elimination. *SIGCOMM Comput. Commun. Rev. 38*, 4 (Aug. 2008), 219–230.

[15] Benson, T., Akella, A., Shaikh, A., and Sahu, S. Cloudnaas: A cloud networking platform for enterprise applications. In *Proc. of ACM SoCC* (Cascais, Portugal, Oct. 2011).

[16] Dixon, C., Krishnamurthy, A., and Anderson, T. An end to the middle. In *Proc. of USENIX HotOS* (Monte Verità, Switzerland, May 2009).

[17] Handley, M., Paxson, V., and Kreibich, C. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *Proceedings of the 10th conference on USENIX Security Symposium* (2001).

[18] Joseph, D., and Stoica, I. Modeling middleboxes. *Network, IEEE 22*, 5 (September 2008), 20–25.

[19] Rajagopalan, S., Williams, D., Jamjoom, H., and Warfield, A. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proc. of USENIX NSDI* (Lombard, IL, Apr. 2013).

[20] Roesch, M. Snort - Lightweight Intrusion Detection for Networks. In *Proc. of USENIX LISA* (Nov. 1999).

[21] Sekar, V., Egi, N., Ratnasamy, S., Reiter, M. K., and Shi, G. Design and Implementation of a Consolidated Middlebox Architecture. In *Proc. of USENIX NSDI* (San Jose, CA, Apr. 2012).

[22] Sherry, J., Hasan, S., Scott, C., Krishnamurthy, A., Ratnasamy, S., and Sekar, V. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proc. of ACM SIGCOMM* (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 13–24.