Virtual Middlebox Management for Cloud

Peter Feifan Chen and Nodir Kodirov
Class project final report for 538B: Distributed Systems, Computer Science, University of British Columbia

I. Introduction

Middleboxes perform a wide range of functions in a network. Popular examples include load-balancing, NATing, firewalling, caching, IPS, tunneling, redundancy elimination and etc. [12]. The importance of middleboxes in enterprise deployments was shown in [22], where it was found that in a typical network the number of middleboxes rivaled the number of L2/L3 switches. However, today, the deployment of middleboxes within a network remains challenging. Works such as [22] and [17] show that middlebox configuration errors are extremely prevalent, that physical middlebox fail-overs are ineffective and that middleboxes incur high capital and operational expenses.

There are multitudes of reasons for this. First, the proprietary and vendor-specific nature of middleboxes require per-middlebox and per-vendor expertise and require network administrators to manually reason about the middlebox's placement and the network forwarding rules on the middlebox path. Second, middleboxes must always be overprovisioned to handle a peak load. This leads to large inefficiencies in terms of both capital and operating expenditure. These limitations led to the virtualization of the hardware-based middleboxes.

Although virtualized middleboxes resolve some problems, they remain challenging to deploy in large-scale cloud deployments and must meet the needs of the endpoints they serve. For example, VM-based endpoint servers are extremely popular way of deploying cloud-services for performance and availability, in a similar manner, middlebox deployments need to converge with the endpoints they serve, i.e., middleboxes need to be scalable and available as well (in the rest of the work, we refer to *virtual middleboxes* as just *middleboxes*).

In order to overcome these issues, there has been a calling for a unified control mechanism for middleboxes in a similar manner to SDN controllers for L2/L3 switches. Like SDN controllers, the unified control system will abstract away the low-level configuration to more tractable high-level declarations. Further, the unified control mechanism should be able to react dynamically to a load change through scaling or replicating middleboxes to meet both performance and availability demands.

Unlike OpenFlow-based L2/L3 switches that can be controlled with a centralized SDN controller [13], middleboxes are usually proprietary, perform wide range of functions and maintain shared state with varying data structures. Further, middlebox functions inherently lack the end-to-end property, which results in ambiguities regarding where the next hop is when middleboxes are chained together. Finally, unlike OpenFlow, which makes control decisions on a flow granularity, some middlebox states need to be updated on a per-packet basis, making the indirection to the centralized controller costly.

These aspects of middleboxes make building a unified controller for middlebox deployments tantalizingly difficult. In this course project, we take a preliminary stab at this complex problem of building a unified controller for middleboxes. Our goal is to design a solution for building scalable middleboxes for cloud deployments. We designed our solution for the increasingly popular container-based cloud setting and leveraged abstractions provided by a container cluster manager for middlebox management. We show that our implementation has the following properties:

- our middleboxes are dynamically scalable
- our middleboxes' shared states consistency is customizable; one needs to trade it off with performance depending on the consistency requirements of the state
- our solution can be generalized to many middleboxes and their states
- our solution is deployable by cloud providers.

We will describe challenges associated with middlebox management in general, and with container-based cloud deployments in particular in the Section II. Our approach, system implementation and evaluations are provided in Section III and Section IV respectively. We discuss the generalizability of our model in Section V and related work in Section VI. Finally, we list interesting future work directions and conclude in Section VII and Section VIII.

II. Challenges

There are multiple challenges associated with middlebox management. They can be divided into several categories, such as configuration, placement and composition, scaling and consolidation, and shared state management. Subsection II-A describes each category in more detail. It will also simultaneously describe the functionalities that the middlebox controller is supposed to be able to handle. Then, we state which of these challenges we are interested to handle in this course project. Subsections II-B and etcd will describe how these challenges apply to the container context. In particular, we describe how Kubernetes' abstractions can be leveraged to address the issues we are interested in.

A. Development Challenges

Configuration, varies by middlebox and vendor type. The purpose of a middlebox controller is to present a higher-level abstraction where the network administrators specify only a policy (e.g., "drop all connections that exceed 100 KB/s in throughput" for a firewall or "send all traffic in a specific subnet to be processed by an IDS, then by a WAN optimizer"). The higher level policy is translated to low-level middlebox-dependent configuration. This reduces operational expenses by

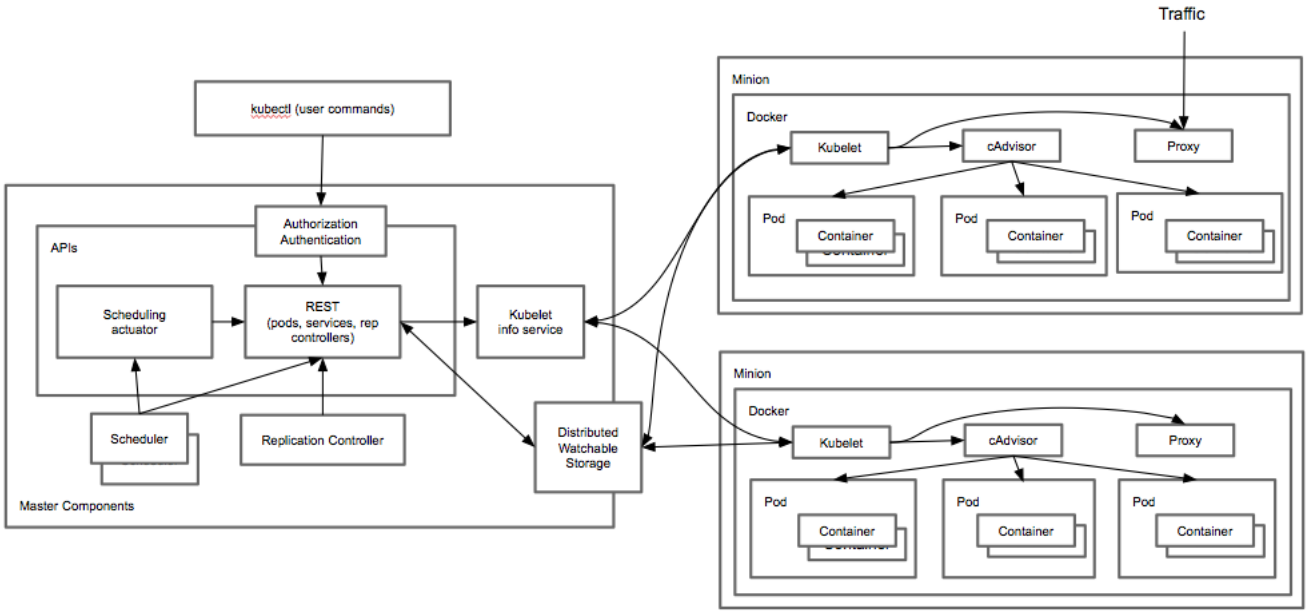

Fig. 1. Kubernetes master and minion nodes, and relationship between different modules of the cluster manager.

reducing the amount of specialized expertise required. The challenge is the variety of middleboxes and their functions.

Placement and Composition, of middleboxes traditionally requires careful and manual reasoning by the system administrator in order to interpose on the network path. A middlebox controller should be able to make the placement of middleboxes in a topology-independent manner by re-writing L2/L3 switch rules based on high-level policy. The challenge here is the next-hop ambiguity caused by chaining middleboxes together, mangling of headers caused by middlebox internals, limited TCAM space for the variety of rules arising from middlebox combinations and optimal co-location of the chained middleboxes.

Scaling and Consolidation, is not possible for physical middleboxes. However, the rise of VM-based middlebox solutions enables the capability to scale-in and scale-out the number of middleboxes on demand. Also, different middleboxes can be consolidated on a single physical machine. Consolidation could also potentially enable performance improvements by improving co-location and reuse for chained middleboxes (e.g., decode lower layers only once for two different application-level middleboxes). The middlebox controller would dynamically scale middleboxes based on demand and consolidate middleboxes based on its global view. The challenges are overcoming proprietary nature of middleboxes for consolidation and managing the shared state of middleboxes for scalability.

Shared state management, to improve the availability of services, middleboxes failures needs to be handled by the middlebox controller. This requires the controller to detect failure and re-route flows to another middlebox replica that is up-to-date with the last known state of the failed box. The challenge is similar to scaling, where there needs to be a way to manage shared middlebox states.

In this work, we are interested in handling scaling and shared state management. We do not address configuration, placement, composition and consolidation as these challenges have more ties to other areas than Distributed Systems. For example, middlebox configuration is more about developing policy abstractions, perhaps with a domain specific language, which points us in the programming languages domain [4]. Placement, composition and consolidation are optimization problems[21]. Although these challenges are interesting on their own, we focus on scalability and shared state, which are more related to this course.

B. Kubernetes for middlebox management

We decided to work with middleboxes in a container environment as containers are increasingly deployed in the data center environment. To increase applicability, we designed our solution in the context of Google's Kubernetes cluster manager [10], which is a widely used solution in large-scale clouds.

Kubernetes acts as an orchestrator for large-scale container deployments. Figure 1[1] shows its system diagram where Kubernetes is responsible for centrally creating, destroying and interconnecting containers across multiple physical machines.

Kubernetes itself is composed of three powerful abstractions: *pod*, *replication controller* and *services*. *Pod* groups related container together, and is the smallest computational unit managed by Kubernetes. *Replication controllers* monitor the number of pod replicas running on the system and ensure that a given number of pods are always running. It creates (or deletes) additional pods if some pods fail during runtime. Pods are ephemeral in nature (created, relocated or deleted). In order to deal with pods that come and go, Kubernetes uses *services* abstraction which is a name-based discovery mechanism for pods. In this way, locating where the pods are is abstracted

[1]Figure credit http://www.centurylinklabs.com/what-is-kubernetes-and-how-to-use-it/

away from clients, who can access them directly via service name. Figure 2[2] shows the relationship between the different Kubernetes abstractions.

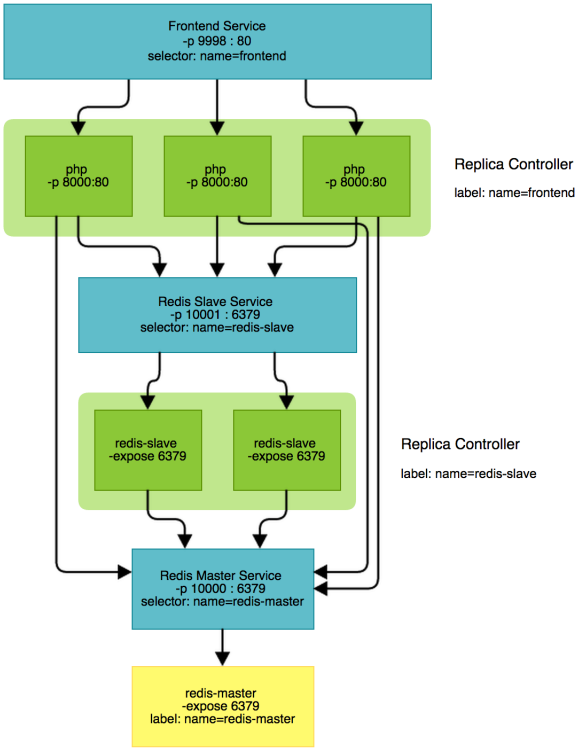

Fig. 2. Relationship between Kubernetes components. It shows relationship of pods, replication controller, and services in Guestbook application. Here *frontend service*, *redis slave service*, and *redis master service* are Kubernetes services connecting *frontend* and *redis-slave* pods. Both of these pods are managed by replication controller. *redis-master* is a independently running pod not managed by any replication controller.

These abstractions not only make Kubernetes well-suited for large-scale container management, but also allows us to architect wide range of middleboxes, as we will describe in more detail in Section III.

C. Etcd for shared state management

Kubernetes stores cluster management related meta-data information and state in a distributed key-value store called etcd [3]. Etcd exposes UNIX file system like structure for key-value operations and provides a REST API to perform a set of operations such as get, put, delete, test-and-set and watch. Puts can have an attached time-to-live. Watch allows an application to be informed of changes to a value via long-polling. Test-and-set can be based on value or modified index.

Etcd uses Raft consensus algorithm to coordinate nodes. It is designed to work with cluster sizes of between 3 to 9 nodes. Heartbeats are regularly sent out by a leader node to all follower nodes to declare liveliness. A separate randomized election time-out is kept by all follower nodes to detect leader failures and begin re-election. Followers become leaders once they receive a majority of votes. On each leader election, a term number is monotonically increased to prevent multiple leaders, although multiple leaders can still exist transiently.

All puts are strongly consistent as they are appended to the Raft log (requires majority) and goes through the leader node. Each write increments a modified index. This index provides a means of versioning such that applications never miss an update, e.g., applications can get a key of a specific index value. Gets have three different consistency semantics: any, consistent and quorum. An *any* read is eventually consistent, but is fast, as it can read from any node. A *consistent* read must read from the leader and therefore is strongly consistent only from the viewpoint of operations of a particular node. Finally, a *quorum* read is strongly consistent from the viewpoint for all nodes, but is the most expensive, as it involves an extra round-trip between the leader and all followers to ensure that another leader was not elected before returning the read value.

The log entries in etcd are compacted periodically through snapshots written to stable storage.

III. Implementation

This section describes the implementation and operational details of our sample rate-limiting firewall middlebox. We will discuss the issue of shared state, and the rationale behind using etcd for shared state management. We will also describe the environment we use for our experiments. Our source code and all experimental data is accessible online [2]; forked repository from original Kubernetes source base [10].

A. Sample middlebox

A middlebox is composed of two basic components: the processing logic and the state. In our case, the processing logic functions as a rate-limiting firewall. We run this firewall as an application inside a container. The container is run inside of a Kubernetes pod. This design allows us to support a wide range of middleboxes such as DPI, transcoder, and IDS by changing the application in the container.

Figure 3 shows overall design of our firewall. For simplicity, we experimented with an echo client and echo server as our endpoints.

The firewall has a per-source IP counter that is incremented on every packet and reset to zero periodically. When the number of packets sent by a particular source IP address rises above a predefined threshold, all packets for that source IP are dropped until the count is reset. For example, if we need to rate-limit all connections to 100 packets per second, our threshold value will be set to 100 and counter reset frequency set to one second.

The shared state in our firewall is the threshold for dropping packets. We push this threshold for dropping packets into etcd (100 in the above example). When a new replica is created, it splits this threshold evenly between all existing replicas. This is done by the firewall application itself. When a new firewall starts, it checks how many firewall instances are already running, calculates the new even-split threshold and updates all the firewalls' threshold values in etcd. For example, when a second firewall is created, it will check etcd and become aware of the first firewall instance with threshold equal to 100. The second firewall then assigns 50 to its own

[2]Figure credit http://allthingsopen.com/

threshold and also updates the first firewall instance's threshold to 50 in etcd. The first firewall instance learns of the new threshold when it next performs a get operation on the value in etcd. Our firewall supports performing a get operation to check for new thresholds at a fixed frequency interval or on a per-packet basis.

The reason for an equal split of the counter is that Kubernetes' service abstraction uses a round-robin scheduler, which guarantees equal division of the flows between firewall instances.

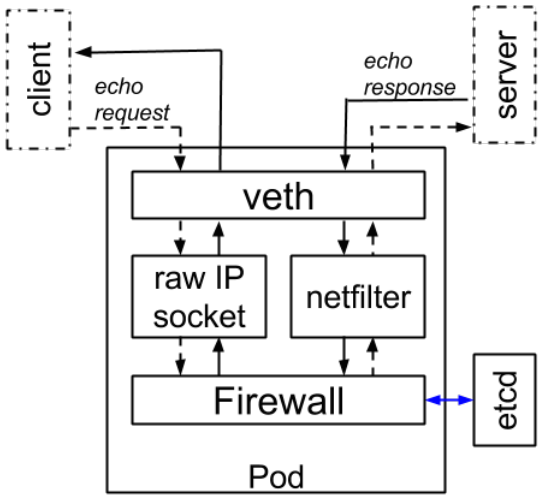

Fig. 3. Rate-limiting firewall system diagram. Dashed lines show echo request packets flowing from client to server. Solid lines show response packets with reverse direction.

The firewall uses Linux kernel's netfilter and iptables to enforce rate-limiting rules. Netfilter is used to capture inbound packets by reading from an iptable queue that captures all Rx traffic except those of etcd. Packet processing is performed by Golang's gopacket library [9]. We modify the source and destination IP addresses of the packet in order to redirect it between an echo client and echo server. We keep track of mappings between connection endpoints by using the unique source port used by the Kubernetes service redirection. The reason we have to modify the packet is because the Kubernetes' service abstraction performs transport layer redirection by opening two separate TCP connections. Because we are using the service abstraction, the second TCP connection actually has the firewall as destination rather than the echo-server. By using Kubernetes service as a redirection mechanism and modifying packet headers, our firewall is able to interpose on the traffic between echo client and echo server (see Figure 4 for illustration of this interposition).

B. Shared state management

Pods are meant to be ephemeral in nature.Therefore placing state in the middlebox would require the ability to migrate it when new replicas are created or old ones destroyed. This is currently impossible as replication controller does not expose these events to the outside world. Therefore, we decided to push shared state out of the middlebox to a shared distributed store, etcd. We also considered a pod-local etcd. However, the ephemeral nature of pods again made this impossible as etcd is designed to operate with a quorum in order to support rejoins of failed etcd nodes. Therefore, there has to be a quorum of previously active pods. This condition is impossible to satisfy and runs opposite to the semantics of scalability, so we opted for pod-external etcd.

C. Experimental Set-up

Figure 4 illustrates general model of our set-up. Routing between pod-based middleboxes and endpoint servers is accomplished through the service abstraction. As it is shown in Figure 2, all pods in Kubernetes are assigned a label. All replicas of the same pod have the same label. Kubernetes services keep mapping of pods, their labels and IP address of these labeled pods. One can address such pods by name through Kubernetes services which then redirects the connection to a random pod with that label. We use services to route traffic to and from our middlebox pod.

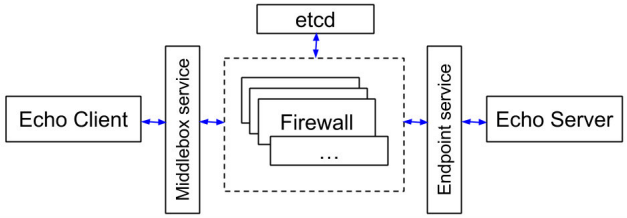

Fig. 4. Firewall pods interpose on the connection between echo client and server. Number of firewall instances are controlled by Kubernetes replication controller and firewall instances periodically fetch new shared state from etcd.

In order to support dynamic scaling, we instrumented Kubernetes with custom monitoring module. This module uses statistics reported by the cAdvisor [8] to collect each middlebox container's CPU, RAM and network bandwidth usage. Once usage crosses a predefined threshold, the monitoring module creates an additional firewall instance by changing the number of replicas maintained by the replication controller. Although not illustrated in the Figure 1 this monitoring module runs as one of the Master components (round boxes) and interacts with the replication controller to change the number of firewall replicas.

IV. Evaluation

We carried out a number of measurements to evaluate how well our system is able to address challenges described in Section II. We were mainly interested in scalability and shared state management of the middlebox controller. This section measures that our firewall pods correctly enforce rate-limiting policy when scaled, that we can scale middleboxes dynamically based on a specified resource limitation and the overhead imposed by pushing state out of the middlebox.

A. Correctness

We first verify if our firewall is correctly enforcing the policy of limiting throughput to 100 packets per second regardless of the number of firewall instances. We measure the Rx rate at the echo server. We run the experiment with one, two and four firewall instances as illustrated in Figure 5.

In this case, a single echo client sent increasing number of TCP packets containing a single ḧelloẅord to the echo server (see Figure 4 for packet flow). In all experiments, one ḧellom̈essage counts as one packet. Figure 5 shows that the number of packets client sends and server receives rises linearly until the the threshold of close to 100 packets/second is reached, at which point the threshold plateaus. This remained true with two and four firewall instances, which confirms our firewall is able to correctly enforce rate limiting policy when scaled. Small jitter around 100 packets/second is due to our imperfect packet counting method as we rely on perfectly synchronized clocks on client and server nodes.

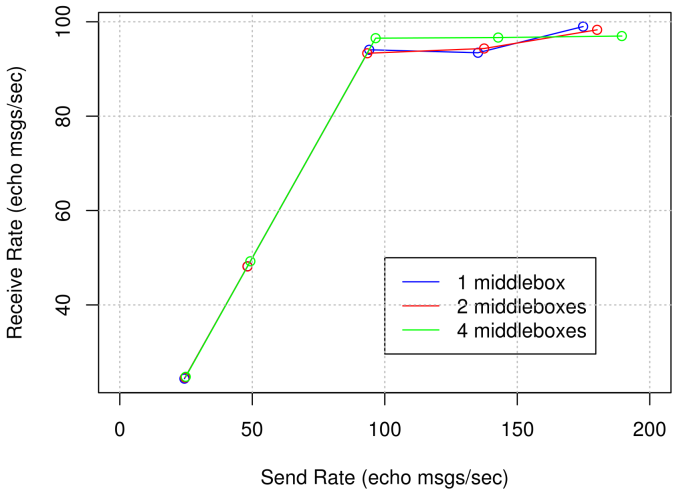

Fig. 5. Correct firewall functionality. Single echo server's Rx is rate-limited to 100 packets per second as expected.

B. Monitoring based Scaling

We also verify that our monitoring module is able to add new firewall instances when a particular resource utilization reaches a predefined threshold. As a threshold we decided to use a RAM utilization of 60%, i.e., when a particular firewall reaches this threshold, the monitoring module is expected to create an additional instance while maintaining correct rate-limiting policy. As previous subsection already confirmed our firewalls' ability to maintain correct rate-limiting when multiple instances are run in parallel, this subsection will focus on the threshold detection and firewall instantiation.

For this experiment we assign 70MB RAM to each firewall pod. We start with only one pod. No load is generated during the first 30 seconds, after which the echo client starts increasing the packet rate. Figure 6 shows RAM utilization of each middlebox over time as we increase the packet rate. It shows the second firewall instance is created when RAM utilization of the first firewall instance hits 60%. After a period of time, the second firewall instance also reaches 60% threshold (s̃0 seconds after) which causes the third firewall instance to be created and so on. Thus, we were able to confirm that our monitoring module is able to dynamically scale middleboxes in response to load.

Note that in this experiment our monitoring module checked pod RAM utilization once per second. It is possible

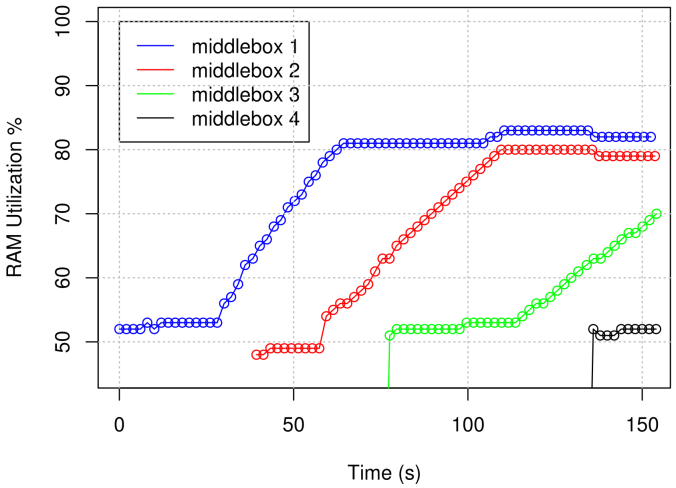

Fig. 6. Dynamic scaling up of the firewall instances. Additional pods are created when RAM utilization of any firewall container reaches predefined 60% threshold. Blue line (leftmost) represents RAM utilization of the first firewall, red line (second left) represents the second firewall and etc.

to make monitoring finer at the expense of higher monitoring overhead. However we believe one second interval was reasonable for our experiments.

Although scaling up firewalls worked correctly, we were not able to verify scale down feature of the monitoring module due to limited statistics reporting support provided by the cAdvisor in Kubernetes at the pod level [11]. In particular, another problem with the scaling down experiment is that cAdvisor does not report decreases in pod's RAM utilization. This means that the RAM utilization stays constant or goes higher, which restricts us to scale up experiments only. However, this is not a fundamental limitation of Kubernetes, but instead is only an unsupported feature that will be rectified in future releases. Once cAdvisor and Kubernetes can report correct statistics, we should be able to run scaling down experiments without additional effort.

Moreover, our monitoring module also supports CPU and network bandwidth based scale up and scale down. These features are also not fully functional due to limitation of Kubernetes and cAdvisor. Once these limitations are eliminated, our monitoring module should also support these metrics.

C. Shared-state Overhead

Middleboxes can trade off performance with consistency. We are interested in empirically evaluating the different overhead costs of our firewall with different consistency requirements for our setup. We measure the time required to process each packet in the firewall. Figure 7 shows latency for four different firewall implementations. In all the experiments, a single echo client sends 10K packets at 1000 packets/second. To measure the latency we take difference between time packet is sent from the client and received by the server. Figure 7 shows min, max and mean processing time for 10K packets.

The first one is the base case where firewall does not contain shared state and has an infinite threshold. Therefore, we expect the base to experience the smallest latency. In our case each packet has a latency of 0.3 ms on average.

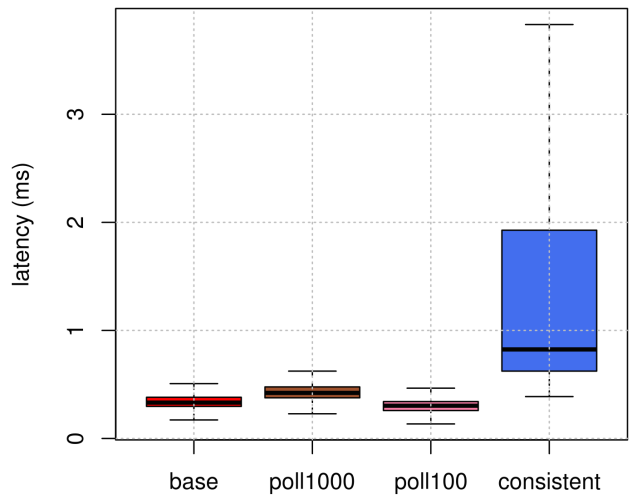

Fig. 7. Etcd overhead on four different implementations of the firewall. Here *base* is expected to impose the lowest latency and *consistent* is the highest.

Poll1000 and poll100 experiments update etcd state at 1000 ms and 100 ms intervals respectively. Both of these implementations impose similar overheads as the base case. We expected poll100 to impose more overhead compared to poll1000, but this is not reflected in the figure. A possible explanation is that the cost of polling is amortized over hundreds of packets, even at 100 ms polling intervals, thus the extra overhead incurred cannot be measured within the precision of our timing mechanism. However, for the case of a consistent firewall, where shared state in etcd is fetched for each packet, we see that the latency is drastically higher (3x) than the base case, as expected.

This experiment allowed us to evaluate the overhead pushing shared state out to etcd. For different type of middleboxes, middlebox operator needs to carefully consider the trade off between correctness and performance. For our case this trade-off is determined by the guarantee that this middlebox is expected to provide to the network.

V. Discussion

We believe our solution generalizes to most middleboxes. Kubernetes' pod abstraction can be easily leveraged to wrap most middlebox functionality while being managed by the replication controller. Further, middlebox chains can be constructed via combining middlebox containers into a single pod. This is similar to CoMb [21], with the policy enforcing shim layer replaced by Kubernetes' service abstraction. However, unlike CoMb, the granularity at which middleboxes need to be combined into pods (hyperapp in CoMb) can be changed, which makes the networking orchestration more like [18], where network configuration takes middleboxes into account.

Kubernetes service abstraction allows our pod-based middleboxes to be launched in a topology-independent manner and gives us a means of interposing middleboxes on the traffic flow path. This aspect of Kubernetes is interesting and unique in terms of middlebox placement as the discovery mechanism is achieved by name rather than re-writing router forwarding table rules. As services are aware of new pods created or pods destroyed by the replication controller, this provides a useful substrate for composing and scaling middleboxes.

By pushing middlebox shared state out to etcd, we allow our pod-based middleboxes to contain only local state and thus capable of being scaled independently by the replication controller, as Kubernetes originally intended pods to be. Separating out the shared state also gives the pod-based middleboxes a configurable trade-off between consistency and performance. We believe this model can fit any middlebox as it separates the storage of state and processing logic - enabling the processing logic of the middlebox to be scaled independently. Further performance can be improved by trading correctness at a per-state granularity.

Take the case of a redundancy eliminating (RE) middlebox. RE fingerprints can be stored in etcd for comparison by any of the RE replicas and retrieved for comparison for new flows. For an application-level load-balancer, every new flow can be used as a key whose value is the IP address of the end-point server in order to provide affinity across the replicas.

Thus, we believe our solution is capable of supporting all types of middleboxes and their functionalities.

VI. Related Work

There has been much recent work in the area of virtualizing middleboxes as network functions and defining methods for managing them. We are not alone in envisioning a future of centrally managed and dynamically scalable middleboxes that are easily configured and upgradable, and cost less in both capital and operational expenses. In [6] Gember et al. motivate the need for a unified control plane for middlebox management and outline the issues that make middlebox management unique. It provides a general framework for potential solutions, but does not go into any details.

APLOMB [22] studies the importance of middleboxes in data center and proposes outsourcing middleboxes to the cloud. We believe deploying middlebox-as-a-service is simply one of the possible deployment scenarios that a middlebox controller can support and therefore complementary to our work.

CoMb [21] advocates for consolidating middleboxes and building them from re-usable modules. This provides both performance and cost benefits as it allows controllers to exploit multiplexing multiple middlebox applications on a single machine and chaining related packet-processing layers of different middleboxes together. Management of CoMb boxes is done through a network controller which constructs the hyperapp (chained middlebox apps) and then maps them onto physical machines with available resources. A policy enforcing shim layer is placed on each machine for routing flows between apps within the hyperapps. Provisioning is done by expressing the desired properties of the logical middleboxes as an ILP problem, optimizing for least maximum load or resource usage. VMs and containers were mentioned as possible avenues for deploying these hyperapps. Complementary to CoMb is xOMB [1]. xOMB aims to open up middlebox architecture to build more flexible, programmable and scalable middleboxes on commodity servers. It provides a pipelined model where middlebox functions are individual stages in the pipeline. It abstracts away flow management (connection management,

socket I/O, data buffering) and provides RPC based arbitrary message passing for state management between middleboxes and a controller. Both of these works are complementary to our Kubernetes pod-based middleboxes, as pods can chain multiple middleboxes and Kubernetes' scheduler can be extended to make load-aware pod placement.

Other works such as Split/Merge [19] provide mechanisms to migrate or merge state when middleboxes are scaled up or down. An SDN based flow manager called FreeFlow was used to re-direct traffic to the correct middlebox after scaling. OpenNF [7] takes Split/Merge further by providing a more generalized abstractions for state management. These abstractions enable loss-free and in-order migration of packets for flows that are still active, a known limitation of the Split/Merge. However, neither dealt with non-partitionable shared-state (state involving multiple flows), arguing that they are rare and do not lie on the critical path for decision making. However, redundancy eliminating middleboxes and application-level load-balancers are obvious exceptions to this argument. In this class project we also consider shared state management by pulling state from the middlebox and use a well-known distributed store etcd to maintain the state. Unlike these two works, we don't directly orchestrate the network as Kubernetes' service abstraction already supports flow redirection to newly created (or deleted) middlebox instances (pods in our case).

Finally, placement and composition of middleboxes were explored in [18]. It tagged each packet to eliminate ambiguity on next-hop and tunnels to make TCAM space usage efficient. Authors also built a controller for automatically translating logical topology in the form of A→B→C into SDN rules, making middlebox placement topology-independent. Further, the placement and proportion of traffic handled by a middlebox in the topology was expressed as an ILP problem that is optimized for load and TCAM space. Again, this is complementary to CoMb by eliminating the need for the policy enforcing shim layer and allowing for finer grained middlebox multiplexing by removing the need to deploy chained middleboxes as a single hyperapp in a single VM or container. Stratos [5] pursued a similar vein, but also included a resource controller for multi-staged scaling and also expressed middlebox provisioning as an ILP problem. Therefore, Stratos can be viewed as a synthesis of multiple works mentioned above.

VII. Future Work

This course project can be extended to make middlebox management practical in a container context. We describe future works by dividing them into two categories: first one is informed by the challenges we faced during project development and the second one is informed by our taste of interesting research. We describe each category in separate subsection.

A. Implementation Challenges

1) Kubernetes: Ideally, we will have our own Kubernetes process dedicated to manage middlebox pods as a first-class entities. This will enable bootstrapping of distributed data store, perform dynamic scaling, monitoring and handle middlebox state. Further, the replication controller needs to be instrumented with finer granularity control, such as which middlebox to destroy and which physical machine newly created middlebox should be placed on. Also, the service abstraction is interesting in that it enables middleboxes to be found by name. However, it is insufficient in its current form as it imposes constraints on how middlebox can be implemented (e.g., must be end-to-end) and is not scalable as the number of chained middleboxes increases. Finally, services should encapsulate packets for redirection rather than using a TCP relay. This prevents all source IP addresses to be the same regardless of the client IP address, which is clearly detrimental to many middlebox functionality.

2) Distributed Store: Etcd has its shortcomings. It cannot be used pod-locally for reasons mentioned in section III-B. Also, etcd is not designed for large amount of data. For example, redundancy eliminating middlebox's fingerprints can reach gigabytes which can not be handled well by etcd. In future we should use distributed store without such limitations or study what kind of middleboxes are well-suited for etcd.

3) Heterogeneity: Middleboxes do not necessarily have to be container-based in order to be used in container deployments. They just need to fit within the abstractions provided for container discovery used by the cluster manager (e.g., Kubernetes). The advantage for pod-based middleboxes is that it is easier to co-locate with the endpoints as their management converges at the cluster manager level. However, in future we would like to explore VM-based middleboxes, such as ClickOS for fast packet-level processing [15] and xOMB for flow-level processing [1].

4) Monitoring and Management: The cAdvisor monitoring system works on a container level, not a pod level. Further, we did not touch upon resource allocation, centralized logical configuration, placement, or routing optimization in this course project. There is a large body of literature on collecting monitoring information for wide range of management objectives as we described in section VI. In future, we could also develop middlebox-specific Kubernetes process to enable pod-based middleboxes run their own specific controller which adds even greater flexibility in managing state.

B. Potential Research Directions

First of all, previous works worked on a subset of middlebox management problems. However, unlike in the SDN world[13], [14], there does not seem to be an unified middlebox control plane that researchers can actually use to handle resource allocation, centralized logical configuration, routing, placement and state while at the same time provide availability, scalability and fault-tolerance in the centralized controller itself. It is possible that a recent open-source project called OpenDaylight[16] seeks to fulfill this exact role. Nevertheless, if that is the case, we can still study what OpenDaylight supports and what it is missing in order to extend OpenDaylight to encompass more properties listed above.

We would like to explore advantages of pushing state out of middlebox and reducing middlebox itself to simple decision logic (processing pipeline). This was mentioned in [6], which was dismissive of this idea, claiming that it would be restrictive for middlebox innovation. However, to our knowledge, no follow-up work pursued this path even though pushing state out of the middlebox would lead to a whole new way of building

middleboxes and likely require industry support from the many disparate vendors. Perhaps, we can even go a step further. Similar to how independent network controllers can be built on top of SDN controller NOX [13], maybe each middlebox vendor can also supply a middlebox controller specific to their middlebox that can run on top a NOX-like middlebox central controller. The vendor can decide to push state out to a distributed store or keep them internal to the middlebox. What kind of abstractions the central middlebox controller provides to vendor controllers and benefits of this model remains to be explored. One difference that immediately jumps to mind is that pushing state out enables us to manage middleboxes at per-state granularity (e.g., different states in a middlebox might require different consistency), while current methods [19], [7] operate at flow granularity, despite the fact that they are managing state and not flows.

Finally, a more theoretical line of work similar to [20] can be explored. This work studies ways of performing SDN rule updates such that no packet or flow sees two different configurations of the entire network. It formalizes this into a verification problem that can be checked with a model checker. Benefits include allowing programmers to reason about network properties from static configurations as packets or flows will never see a transient one. One could explore how such rule update can be applied to scenarios where a middlebox configuration change is required when multiple middleboxes are chained? Furthermore, what if this middlebox composition changes flow headers or performs stateful processing? These questions are important as chained middleboxes might require no flows to ever see a transient set of middlebox configurations.

VIII. Conclusion

In this course project we studied challenges of middlebox management in a container context. In particular, we designed, implemented and evaluated system which leverages container cluster manager (Kubernetes) to address scalability and shared state management of the middleboxes. We developed monitoring module to keep track of middlebox resource usage and spawn additional middleboxes when predefined threshold usage is reached. This module leverages Kubernetes' replication controller abstraction to achieve dynamic scaling. We also experimented with middlebox shared state by pushing it to etcd and observed consistency and performance trade-offs.

In general, Kubernetes is well-suited for middlebox management as it provides essential abstractions for middlebox management. We reported some limitation of the Kubernetes' current implementation and how it can be extended to support more flexible middlebox management.

References

[1] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xomb: Extensible open middleboxes with commodity servers. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '12, pages 49–60, New York, NY, USA, 2012. ACM.

[2] P. F. Chen and N. Kodirov. Course project source code. https://github.com/knodir/kubernetes/tree/master/VMC. [Online; accessed April-23-2015].

[3] CoreOS. coreos/etcd. https://github.com/coreos/etcd. [Online; accessed April-18-2015].

[4] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 279–291, New York, NY, USA, 2011. ACM.

[5] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar. Stratos: A network-aware orchestration layer for middleboxes in the cloud. Technical report, Technical Report, 2013.

[6] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella. Toward software-defined middlebox networking. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pages 7–12, New York, NY, USA, 2012. ACM.

[7] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. Opennf: Enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 163–174, New York, NY, USA, 2014. ACM.

[8] Google. google/cadvisor. https://github.com/google/cadvisor. [Online; accessed April-18-2015].

[9] Google. Packet library for golang. https://godoc.org/code.google.com/p/gopacket. [Online; accessed April-21-2015].

[10] GoogleCloudPlatform. Container cluster manager. https://github.com/googlecloudplatform/kubernetes. [Online; accessed April-21-2015].

[11] GoogleCloudPlatform. Kubernetes' current limitation on resource monitoring. https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/resources.md. [Online; accessed April-21-2015].

[12] I. N. W. Group. Rfc3234 - middleboxes: Taxonomy and issues. https://www.ietf.org/rfc/rfc3234.txt. [Online; accessed April-18-2015].

[13] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: Towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[14] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, volume 10, pages 1–6, 2010.

[15] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473, Seattle, WA, Apr. 2014. USENIX Association.

[16] OpenDaylight. Opendaylight. http://www.opendaylight.org/. [Online; accessed April-21-2015].

[17] R. Potharaju and N. Jain. Demystifying the dark side of the middle: A field study of middlebox failures in datacenters. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 9–22, New York, NY, USA, 2013. ACM.

[18] Z. Qazi, C.-C. Tu, R. Miao, L. Chiang, V. Sekar, and M. Yu. Practical and incremental convergence between sdn and middleboxes. *Open Network Summit, Santa Clara, CA*, 2013.

[19] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 227–240, Lombard, IL, 2013. USENIX.

[20] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 323–334, New York, NY, USA, 2012. ACM.

[21] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 323–336, San Jose, CA, 2012. USENIX.

[22] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 13–24, New York, NY, USA, 2012. ACM.