Yoda: A Highly Available Layer-7 Load Balancer

Rohan Gandhi
Purdue University

Y. Charlie Hu
Purdue University

Ming Zhang
Microsoft Research

Abstract

Layer-7 load balancing is a foundational building block of online services. The lack of offerings from major public cloud providers have left online services to build their own load balancers (LB), or use third-party LB design such as HAProxy. The key problem with such proxy-based design is each proxy instance is a *single point of failure*, as upon its failure, the TCP flow state for the connections with the client and server is lost which breaks the user flows. This significantly affects user experience and online services revenue.

In this paper, we present YODA, a highly available, scalable and low-latency L7-LB-as-a-service in a public cloud. YODA is based on two design principles we propose for achieving high availability of a L7 LB: decoupling the flow state from the LB instances and storing it in a persistent storage, and leveraging the L4 LB service to enable each L7 LB instance to use the virtual IP in interacting with both the client and the server (called front-and-back indirection). Our evaluation of YODA prototype on a 60-VM testbed in Windows Azure shows the overhead of decoupling TCP state into a persistent storage is very low (<1 msec), and YODA maintains all flows during LB instance failures, addition, removal, as well as user policy updates. Our simulation driven by a one-day trace from production online services show that compared to using YODA by each tenant, YODA-as-a-service reduces L7 LB instance cost for the tenants by 3.7x while providing 4x more redundancy.

Categories and Subject Descriptors 500 [*Computer systems organization*]: Availability; 300 [*Computer systems organization*]: Reliability

Keywords Cloud computing; datacenter; load balancer; availability

EuroSys '16, April 18-21, 2016, London, United Kingdom

ACM 978-1-4503-4240-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2901318.2901352

1. Introduction

Modern cloud service infrastructures host multiple tenants running many types of applications, while providing very high uptime SLA [11, 12]. An online service in cloud typically scales by running on multiple servers, each with an individual IP, and receives traffic from outside the service boundary by exposing one or a few virtual IP addresses (VIP). A multi-tenant load balancer service is a critical component of such multi-tenant cloud environments – it receives the traffic destined for a VIP, splits it among the servers assigned to that VIP, and routes it to the individual servers.

As the load balancer touches every packet received by many of the services hosted in the cloud, the performance and availability of online services directly depend on the load balancer, which adds stringent performance and uptime requirement on the load balancer.

Layer 4 (L4) load balancing is a basic load balancing mechanism supported in today's public cloud [18, 20, 25, 38, 48]. An L4 load balancer (LB) splits the traffic coming to a VIP across the servers for that VIP using the L4 fields, *i.e.,* IP tuple consisting of IP type, source and destination IP addresses and port numbers. As online services grow complex, they require content based switching, *i.e.,* the traffic on the same VIP is split across different sets of servers based on the HTTP content, which cannot be done by an L4 LB. L7 LB enables such fine-grained content-based switching, and is a vital building block of many online services [5].

Traditional L7 load balancers are dedicated hardware middleboxes [1, 4] that are expensive and typically only used in private enterprises or private cloud. In the mean time, public cloud providers only offer L4 load balancing as a service, which cannot be easily extended to an L7 LB.

As a result, online services in public cloud have to deploy and manage their own L7 LB solutions (such as HAProxy [5]) that can run on VMs. However, current L7 LB designs are plagued with two problems that severely hamper online service operations, maintenance and importantly user experience. First, the L7 LB designs exhibit limited availability where an LB instance failure results in user-visible connection resets of all connections handled by the failed LB instance. The key reason is that under the proxy-like LB mechanism, for a client request, each LB instance first establishes a TCP connection with the client to get the HTTP

request, then selects the server and starts a new connection with the server. When that LB instance fails, the state of the two TCP connections constituting the *end-to-end flow* is lost; a different LB instance cannot uncover the state in order to take over the flow. As a result, client flows break which significantly hampers user experience and online service revenue. Second, each online service operator has to ensure that it deploys enough L7 LB capacity to handle dynamic traffic demand which can incur high cost.

In this paper, we propose YODA, which enables L7 load balancing as a service to the tenants in public clouds. YODA is based on a scale-out design and uses existing VMs in the cloud to address the above drawbacks of current per-tenant, proxy-based L7 LB solutions. YODA addresses the availability challenge using two ideas. We observe that in current L7 LB solutions, the key reason a client flow handled by a failed L7 LB instance could not migrate to another instance is that the failed instance had end-to-end connections with the server/client, and hence the TCP states for the two connections stored locally at the L7 LB instance would be lost upon the instance failure. The first key idea of YODA is to have the L7 LB instances use the VIP in connections with the server and the client so that neither connection is tied to the instance's IP. This is a prerequisite for migrating the client flow to another LB instance transparently to the client and the server. Our insight is that LB instances can easily use the VIP by leveraging existing L4 LB service in the cloud.

The above mechanism allows the LB instance to receive and forward packets between the client and the server using the VIP, but the *flow state*, consisting of the two TCP states and the selected server, are still stored locally on the LB instance and hence can be lost upon the instance failure. Our second idea is to decouple such flow state from the specific LB instance handling the connection and store it in a high performance persistent datastore, called TCPStore, that we built on top of Memcached [8]. As a result, when an LB instance fails, the flow state of the traffic it was handling can be retrieved from TCPStore by any of the remaining LB instances. Together with the first idea of using VIP, the new LB instance can seamlessly take over the flows. Using VIP and decoupling and sharing the flow state among LB instances this way also simplify providing LB scalability, since the flows can easily migrate to go through other LB instances as LB instances are added or removed.

The second challenge is how to assign the LB rules based on the user policies for all the tenants to the LB instances. Assigning rules for all tenants to all LB instances provides high robustness, where upon one LB instance failure, any of the remaining LB instances can handle its traffic. However, this choice also increases the number of rules on individual LB instances which significantly inflates latency. To address this challenge, we develop a many-to-many VIP assignment algorithm that minimizes the LB instance cost while giving a level of guarantee on both latency and failure resilience.

We evaluate YODA using a prototype deployed on a 60-VM testbed in Windows Azure, and large scale simulations. Our results show that the flow state can be captured, and used on different VMs to maintain the flows, transparently to the servers and clients, and the overhead of decoupling TCP state is very small ($<$1 msec). Our simulation results using a one-day trace from production services show that YODA-as-a-service reduces L7 LB instance cost for the tenants by 3.7x while providing 4x more redundancy.

In summary, we make three contributions: (1) We present the design and implementation of a highly available and scalable L7 load balancer as-a-service for public clouds. (2) We present two key ideas for achieving high availability of a L7 LB: the design principle of decoupling the flow state from the LB instances and storing it in a persistent storage, and leveraging the L4 LB service to enable each L7 LB instance to use the VIP in interacting with both the client and the server. We believe the design principle and the mechanism can be applied to designing other L7 middlebox functions. (3) We present an effective algorithm for calculating VIP-LB instance assignment to minimize the cost while guaranteeing a given level of LB robustness and performance.

2. Background and Motivation

We start with a brief background on load balancing in the cloud, describe the current L7 LB solution HAProxy, and point out its limitations.

Load balancing: An online service typically comprises of many servers that work together as a single entity. Each server in the set has a unique IP address. Each service exposes one or more virtual IP (VIP) outside the service boundary. A load balancer forwards the traffic destined to a VIP to one of servers for that VIP. The indirection via the VIP provides several important benefits. Most importantly, it hides the dynamics within the service from outside users, where individual servers can be maintained or upgraded without affecting users.

L4 load balancing is a basic load balancing mechanism [25, 38, 48] . Cloud-scale L4 LBs such as Ananta [38] and Duet [20] select the server by calculating the hash over the L4 fields (the 5-tuple consisting of IP type, source and destination IP addresses and port numbers), and use IP-in-IP encapsulation to forward the traffic to the selected server.

2.1 L7 load balancer

As online services grow complex, they raise the need for partitioning website logic and data across different sets of servers or even different DCs. For example, an online service can have one set of servers to handle PHP content and a different set of servers to handle CSS content.

L7 load balancing enables such fine-grained partitioning of traffic handled by the online service. In contrast to an L4 LB, an L7 LB provides content-based switching, where the

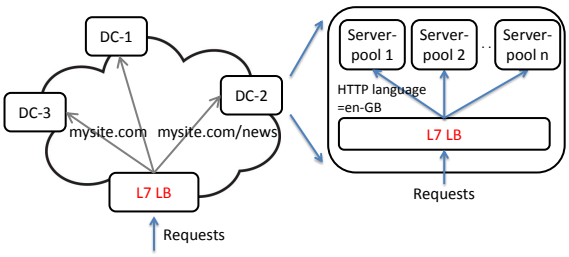


Figure 1. Typical L7 LB deployment.

LB inspects L7 header (HTTP) content in the incoming requests to select the server where the requests are forwarded.

Figure 1 shows the typical usage of an L7 LB by online services as well as CDNs. First, at the edge, the L7 LB is used to select the DC when not all the DCs store all the data and applications (*e.g.,* the requests for `mysite.com/news` is served only through DC2). Additionally, within a single DC, an L7 LB is used to load balance the traffic across different server pools (*e.g.,* the requests with language en-GB is served only through serverpool-1 in Figure 1).

Key requirements: As an L7 LB touches every packet received by online services, its performance and robustness directly affects the performance and user experience of the online services. For this reason, the LB faces stringent robustness and performance requirements. (1) **Availability:** The LB needs to be online, and maintain connections during failures, (2) **Scalability:** The LB needs to adapt to the dynamic traffic load; (3) **Low latency:** The LB should incur minimal extra delay while load balancing the traffic. (4) **Flexible and expressive interface** so online service operators can easily express policies.

2.2 Existing L7 LB

Today, major cloud service provides offer highly available and scalable L4 LB services but not an L7 LB service. As a result, tenants are forced to build their own L7 LB or to use third-party L7 solutions [2, 5] in cloud.

Current solutions in public cloud such as HAProxy [5] enable L7 load balancing using a proxy-like mechanism. First, each proxy LB instance establishes a TCP connection with the client and receives the HTTP content. Next, it inspects the HTTP content and selects a server based on the user policies. Once the server is selected, it establishes a TCP connection with the server and simply copies the data between these two connections. To handle higher traffic load, multiple proxy instances are used, and traffic is split among the multiple proxy instances using DNS or the L4 LB service.

2.3 Limitations of Existing L7 LB

Problem 1 – Low reliability: The current proxy-based L7 LB mechanism faces a key limitation: **Each L7 LB instance becomes a single point of failure**. Since the proxy operates at L4 and establishes TCP connections with both the client and server, when the LB instance fails, the TCP connection state of the connections with the server and client is lost, the packets from the server and client are dropped. Eventually the two connections are terminated after HTTP timeout, which can be several seconds or even minutes. Prior work has also emphasized that low reliability of middleboxes (*e.g.,* LBs) poses a significant challenge to middlebox vendors and network operators. For example, FTMB [43] notes that *many middlebox vendors have argued against resetting the existing connections when failures happen because of the potential for user-visible disruption to applications.*

website	**nytimes**	**reddit**	**stanford**
impact	page timed-out	page timed-out	page timed-out
website	**vimeo**	**soundcloud**	**email service**
impact	service stopped	service stopped	email failed

Table 1. Impact of proxy failure on different websites.

Failures are common: [40] shows the hardware middlebox (common in private cloud) failures have low reliability – cumulatively hardware middleboxes (including LBs, Firewalls) account for 43% of all the network device failures. When the middleboxes fail, all the existing connections on those middleboxes fail, and the traffic is lost. [40] shows that 100's of GB of traffic is lost during LB failures.

Similarly, when the L7 LBs are implemented in the software, *e.g.,* HAProxy (common in public cloud), the connections break when the server crashes. The server crashes occur for a variety of reasons including software bugs, hardware failures, maintenance operations, power failures *etc.*, which are common.

To understand the impact such failures can have on user experience, we emulate a proxy failure that breaks a single established connection, and measure its impact on 10 popular websites in terms of page-load time and session reset. Table 1 summarizes the results[1]. We see that breaking a single established connection due to the proxy failure either elongates the page-load time by 5 min (default Mozilla Firefox browser HTTP timeout) for popular websites such as nytimes.com or reddit.com, or breaks ongoing sessions for websites like vimeo.com or soundcloud.com. Such high latency and session resets can significantly affect the online service user experience and revenue especially for mobile and/or latency sensitive applications [24, 32, 41].

The low reliability of current L7 LBs leave the websites vulnerable when the LBs crash.

Problem 2 – Management overhead: Using current proxy-based L7 LB solutions (HAProxy), online services have to manage their own LB instances, *i.e.,* tenants have to somehow ensure scalability and availability on their own. Online services cannot elastically add/remove LB instances as removing LB instances may break the connections.

[1] Due to page limit, we show the results for 6 websites.

3. Yoda Key Ideas

In the previous section, we saw that the current proxy-based L7 LBs suffers a fundamental limitation: client sessions are broken in case of LB instance failure. In this paper, we propose a new multi-tenant cloud L7 LB service called YODA. Like HAProxy, YODA comprises of multiple instances running on VMs, does not require administrative access to cloud infrastructure (*e.g.,* switches), and hence can be easily deployed by a third- party. But unlike HAProxy, YODA provides high availability and scalability, based on two key observations: (1) Each LB instance should not use its actual IP for connecting with the client or server, since upon failure its IP becomes unreachable[2]. (2) Each LB instance should not store the flow state for the connections to the client/server locally, which will be lost upon failure.

These observations motivate the following design principles in YODA:

Using VIP for client/server connections: In YODA, each YODA instance always uses the VIP while making connections to clients and servers. In other words, the clients and the servers always see the other endpoint of their connections as the VIP. This "front-and-back" indirection masks the YODA instance failures from the clients and servers, and allows for transparent failover of client flows from being handled by one YODA instance to another. YODA uses existing L4 LB service in the cloud to redirect VIP traffic to/from the YODA instances. Specifically, Yoda requires L4 LB to split incoming requests across YODA instances, and SNAT outgoing responses with the VIP. These requirements are easily met by existing L4 LBs [20, 38]. This decoupling of the L4 and L7 LBs provides modularity and enables the design of both LBs to evolve independently.

Storing flow state in persistent datastore: Running YODA instances behind VIP is not enough. When one YODA instance fails, the subsequent packets for a flow handled by the failed instance will be rerouted (by the underlying L4 LB) to another YODA instance. For the flow to be maintained, the subsequent packets between the client and the server should be correctly translated, *i.e.,* the sequence numbers, ports, IP addresses should be consistent with before. To do this, the flow state consisting of the client-VIP and VIP-server connections must survive the YODA instance failure and be accessible and reusable by other YODA instances. To achieve this, YODA decouples the flow state from each instance and stores it in a persistent in-memory datastore, called TCPStore, which can be retrieved by any other L7 YODA instances.

The above ideas for achieving availability effectively allows for transparent client flow migration between YODA instances, and thus naturally provides high scalability, where YODA instances can be dynamically added or removed to match the traffic load without affecting existing flows.

[2] Assigning its IP to another YODA instance potentially in a different part of the DC network can incur a significant delay due to route convergence.

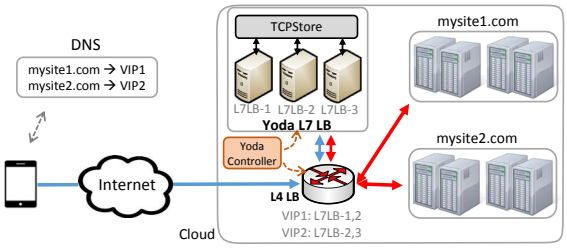


Figure 2. YODA architecture. Red lines denote L7LB-servers traffic. Blue lines denote L7LB-clients traffic.

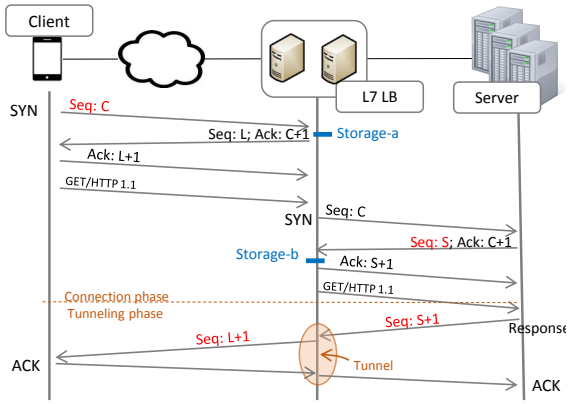


Figure 3. Connection establishment in YODA. The sequence numbers used while establishing the connections are shown over the arrows. The times when YODA instances store flow state are marked in Blue.

Tunneling at L3 for efficiency: A third idea of YODA design is to minimize TCP stack processing and storage operations by tunneling packets. We observe the YODA instance just needs to maintain a TCP connection with the client until it receives the HTTP header, then opens a TCP connection to the selected server and forwards the HTTP request. Since the header has only a few packets, TCP congestion control has not kicked in this phase. After the server receives the HTTP request, the LB instance can simply tunnel the traffic between the server and the client at L3, by properly adjusting the TCP/IP header. Thus in either phase, the LB instance can avoid performing TCP congestion control and leave congestion control to the client and server.

4. Yoda Design

We now describe the YODA design. To simplify the description, we consider HTTP 1.0, where there is one request/response on one TCP connection. The changes required to support HTTP 1.1 are described in §5.2.

4.1 Basic operations

Receiving VIP traffic: Figure 2 shows the YODA architecture. Each online service (mysite1, 2) is assigned a VIP in its DNS mapping, which clients use to send requests. The

requests are first received by the L4 LB in cloud. The L4 LB uses the mapping between the VIP and YODA instances (calculated and set by the YODA controller) to split the incoming VIP traffic among the YODA instances (*e.g.,* VIP1 traffic is split between L7LB1, 2).

YODA **instance**: In a nut shell, an YODA instance performs three functions: (1) *receiving* the HTTP header from the client to select the server; (2) *forwarding* the HTTP request to the server; (3) *forwarding* the data between the client and the server. YODA instances operate in two phases for every flow: (1) connection phase, where it connects with the client and server, (2) tunneling phase, where it forwards traffic between the client and server connections.

Establishing connection with the client: Figure 3 shows how YODA establishes the connection with the client. Upon receiving a SYN packet indicating a new TCP connection, a YODA instance performs two tasks: (1) It stores the TCP header from the client before responding with the SYN-ACK (shown as storage-a in Figure 3), so that other YODA instances can retrieve the TCP fields and the sequence numbers on failure of this YODA instance. (2) It sends a SYN-ACK to the client. For a given connection with the client, all YODA instances generate the same SYN-ACK – the sequence number used for the SYN-ACK is generated by hashing the source IP-port tuple. This avoids the need to store SYN-ACK state in TCPStore. Note that the YODA instance cannot select the server yet as it has not received the HTTP header.

When the HTTP header is received, the YODA instance selects the server based on the user policies. We observe there is no need to send any ACK to the client to receive the HTTP header packets as they typically fit in the TCP initial window (but ACK is sent and added to TCPStore if needed), and there is no need to add anything else to TCPStore in this phase.

Establishing connection with the server: YODA instance establishes a connection with the server using the VIP (using the SNAT functionality of the L4 LB), *i.e.,* the source IP in the SYN packet sent to the server is set to the VIP. When it receives the SYN-ACK from the server, it stores the flow state, *i.e.,* IP addresses/ports and the sequence numbers in the TCPStore before sending an ACK (event storage-b in Figure 3). This ensures the flow state can be recovered by another YODA instance on current instance failure. YODA instances use the same starting sequence number received from the client to establish connection with the server. This reduces packet processing for the subsequent packets.

Tunneling subsequent packets: Since subsequent packets do not need to be inspected, the YODA instance tunnels all subsequent packets between the client and the server on the two connections at L3 as shown in Fig 3. However, the sequence number received in the packet from the server will not match the sequence number used by the YODA instance in its connection with the client, and would require translation to maintain the connection. To do this, the instance keeps the following state locally and also in TCPStore: (1) the starting sequence numbers from the client and the servers C and S, (2) the assigned server.

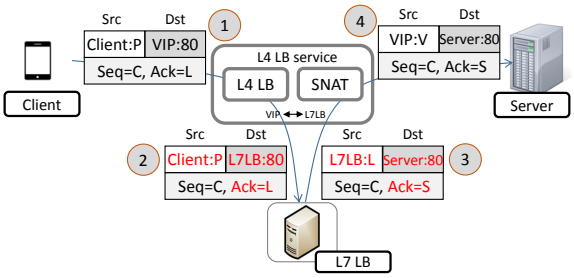


Figure 4. Front-and-back indirection: YODA leverages the L4 LB to establish TCP connections with both the client and the server: both the client and server see the VIP as the other endpoint of their connections.

Figure 4 shows how the source/destination addresses and the sequence numbers in the packets from the client and server are translated, during both the connection phase and the tunneling phase, so that both the client and server will only see VIP as the other endpoint of their connections.

Terminating connection: The flow state stored at the YODA instance and TCPStore is removed when the instance receives FIN-ACK.

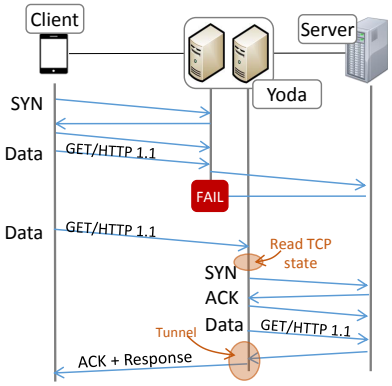
4.2 Handling YODA instance failure

The guiding principle to maintain flows during YODA instance failure is that *each instance stores all the packets it ACKes* (*e.g.,* SYN from client and SYN-ACK from server) in the connection phase in TCPStore, as shown in Figure 3, so that no state is lost on failures. After failure, other YODA instances can retrieve the flow state from the retransmitted packets and from TCPStore. We now detail on how we use this principle to maintain connections.

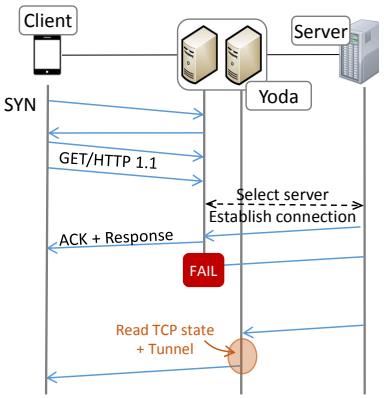
Recall the servers and clients do not see the individual YODA instance failure, as YODA instances use the VIP on the connections with the servers and clients. When a YODA instance fails, the YODA controller detects the failure and removes the failed instance from all the mappings at L4 LB. As a result, the underlying L4 LB forwards subsequent packets from the server and the client to one of the remaining instances. We consider failure during the connection phase and tunneling phase separately.

In the connection phase, if the YODA instance fails before sending SYN-ACK, then when the client resends the SYN packet, the packet is forwarded to one of the remaining YODA instances and treated as the first packet for a new flow (we observe the SYN timeout to be 3 sec in Ubuntu and packets are rerouted by the L4 LB in less than 600 msec, as shown in §7.2).

The more interesting case is when the instance fails any time after inserting the client SYN header into the TCPStore



(a) Connection phase



(b) Tunneling phase

Figure 5. YODA Failure recovery.

and before forwarding the ACK (from server) back to the client as shown in Figure 5(a). As the client does not receive ACK for the HTTP packet(s), it retransmits those packets, which will be forwarded by the L4 LB to one of the remaining YODA instances. The instance looks up TCPStore to retrieve the flow state (SYN header and sequence number), and detects that it is the first data packet. It then starts a new connection with the server, and continues with the Data/ACK exchange with the client. In this way, the client gets the response without knowing the prior YODA instance failed.

Lastly, Figure 5(b) shows the packet flow during YODA failure in the tunneling phase. When the YODA instance fails and another instance receives the re-transmitted packet, it retrieves the flow state from TCPStore, and uses it to adjust the packet header and forwards the packets correctly.

4.3 YODA TCPStore architecture

YODA decouples the flow state from LB instances and stores them in a separate storage, TCPStore. The key requirements for the storage are: (1) low latency, as any latency added by the storage operations inflates the end-to-end latency, (2) persistence, as the flow state is critical to maintaining the client flows during failure.

YODA TCPStore is built on top of Memcached, a scale-out key-value store [8] running on commodity VMs. Memcached provides three APIs: `set(key,value)`, `get(key)`, `delete(key)` for accessing key-value pairsflow state. The key drawback of Memcached is that it stores each key-value pair on a single Memecached instance, and does not provide persistence when a Memcached instance fails. We address this limitation by storing each key-value pair on multiple Memcached instances by modifying the Memcached client library which runs on every YODA instance and handles all Memcached operations. §6 gives more details on the implementation of TCPStore.

The YODA TCPStore minimizes the latency of the storage operations through a series of optimizations including (1) decentralized server selection, (2) concurrently issuing operations to all replica servers, and (3) using long-lived TCP connections between the Memcached clients and servers. We omit the details here due to page limit.

4.4 VIP assignment

So far we have discussed how YODA achieves high availability and scalability – two primary goals of YODA. We now detail the remaining design component on: (1) how YODA selects the server based on the rules, (2) how to assign L7 LB rules for all online services (*i.e.,* VIPs) to the YODA instances to optimize cost, latency, and failure resilience.

Server selection: YODA provides an interface that lets online service operators express their policies as OpenFlow-like rules (§5.1) that consist of match, action and priority fields. The YODA instances match the HTTP headers in incoming new connections against these rules in order to select the server for each client flow.

Since designing a new classification algorithm is not the focus of YODA, it uses existing algorithm from HAProxy with an extension to support priority. HAProxy maintains a single table with all the rules chained, and scans all the rules linearly to select the backend server for every incoming new connection. In YODA, we add a new priority field to the rules, and arrange the rules in the decreasing order of the priority. Priority enables YODA to support rich set of policies easily as detailed in §5.1.

Additionally, YODA maintains a hash table that stores the mapping between the connection (identified using TCP/IP 5-tuple) and the assigned backend server. YODA uses this mapping to forward the subsequent packets on the individual connections.

Next, we detail on how YODA assigns the rules to the YODA instance.

Rule assignment: One simple approach is to assign all the VIPs, and thus all their rules to all YODA instances. We call this scheme all-to-all. Such an approach provides high robustness since any remaining instance can handle traffic for any VIP when some YODA instances fail.

However, it can incur high latency from scanning too many rules for every new flow. We first assess the impact

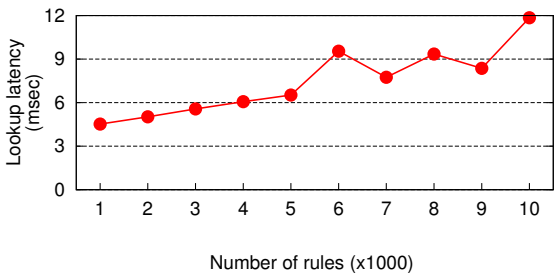


Figure 6. Look-up latency in HAProxy.

Notation	Explanation
Input	
Y, V	Sets of YODA instances and VIPs
t_v, r_v	Total traffic and rules for v-th VIP
o_v	Over-subscription for v-th VIP
T_y, R_y	Total traffic and rule capacity of y-th YODA instance
n_v	# YODA instances assigned to v-th VIP
f_v	# YODA failures for v-th VIP to be tolerated
Output Variable (binary)	
$x_{v,y}$	set if v-th VIP is assigned to y-th YODA instance

Table 2. Notations used in the algorithm.

of the number of rules on latency in HAProxy. Figure 6 shows that the (P90) latency increases about linearly with the number of rules. The latency for scanning 10K rules is roughly 3x than for 1K rules, which is a significant increase especially for latency sensitive applications.

To address these limitations, YODA uses a many-to-many model for VIP assignment, where a VIP (and its rules) is assigned only to a subset of YODA instances. This model significantly reduces the number of rules on each YODA instance which reduces server selection latency.

The YODA controller: (1) calculates the VIP assignment (*i.e.,* which VIPs are assigned to which YODA instances), as the underlying L4 LB splits and forwards traffic at the VIP granularity, all the rules for the individual VIPs are to be installed at the YODA instances where the VIP is assigned. (2) installs the L7 LB rules at the corresponding YODA instances, (3) installs the VIP-to-YODA-instance mapping in the L4 LB, so that VIP traffic is forwarded to the YODA instances that store the L7 rules for those VIPs.

We formulate the VIP assignment problem in YODA as an ILP to minimize the YODA instances for a given latency constraint and level of robustness, as shown in Figure 7.

Input: The input to the ILP includes the (1) set of VIPs (V) and YODA instances (Y), (2) total traffic and rules to each individual VIP (t_v, r_v), (3) traffic and rule capacity for individual YODA instances (T_y, R_y), (4) Number of replicas for individual VIPs (n_v), (5) over-subscription ratio o_v for each VIP (o_v and n_v are specified by online services).

ILP Variable: $x_{v,y}$

Objective: Minimize $\sum_{y \in Y} y_y$, $y_y = 1$ if $\sum_{v \in V} x_{v,y} \geq 1$, else 0

Constraints:

Traffic capacity: $\forall y \in Y, \sum_{v \in V} x_{v,y} \cdot \frac{t_v}{n_v - f_v} \leq T_y$ (1)

Rule capacity: $\forall y \in Y, \sum_{v \in V} x_{v,y} \cdot r_v \leq R_y$ (2)

Number of replica: $\forall v \in V, \sum_{y \in Y} x_{v,y} = n_v$ (3)

Transient traffic: $\forall y \in Y, \sum_{v \in V} z_{v,y} \cdot t_v \leq T_y$ (4)

where $z_{v,y} = max(\frac{x_{v,y}}{n_v}, \frac{x_{v,y}^{old}}{n_v^{old}})$ (5)

Connections migrated: $\sum_{v \in V, y \in Y} m_{v,y} \cdot C_{v,y}^{old} \leq \delta$ (6)

where $m_{v,y} = max(x_{v,y}^{prev} - x_{v,y}, 0)$ (7)

Expressing y_y: $\forall y \in Y, 1 \geq y_y \geq \frac{\sum_{v \in V} x_{v,y}}{N}$ (8)

Expressing $z_{v,y}$: $z_{v,y} \geq \frac{x_{v,y}}{n_v}$ and $z_{v,y} \geq \frac{x_{v,y}^{old}}{n_v^{old}}$ (9)

Expressing $m_{v,y}$: $m_{v,y} \geq 0; m_{v,y} \geq x_{v,y}^{prev} - x_{v,y}$ (10)

Figure 7. ILP formulation.

We calculate $f_v = n_v \cdot o_v$, *i.e.,* the number of YODA instances assigned to v-th VIP whose failure will not overload other YODA instances.

Output: YODA instances assigned to each individual VIP – $x_{v,y}$ is set if v-th VIP is assigned to y-th YODA instance.

Objective: Minimize the total number of YODA instances, which lowers the YODA operating cost.

Constraints:

- **Traffic capacity:** Each YODA instance has enough capacity to handle traffic even after f_v failures (Eq. 1 in Figure 7) for all the VIPs assigned to it. $\frac{t_v}{n_v - f_v}$ denotes the traffic for v-th VIP after f_v failures.
- **Rule capacity:** Each server has enough memory to store the rules for VIPs assigned to it (Eq. 2).
- **Number of YODA instances:** Each VIP is assigned to n_v YODA instances (Eq. 3).
- **VIP migration:** Transient traffic and requests do not overwhelm YODA instances and TCPStore. We detail this constraint in the next section.

4.5 Updating VIP assignment

The VIP assignment calculated at one point of time will not be always optimal, due to traffic dynamics, YODA instance failure and recovery, VIP addition/deletion, and changes in the user policies (different number of rules). YODA adapts to such dynamics by adding/removing YODA instances and re-computing VIP assignment from time-to-time. After computing a new VIP-to-YODA instance assignment, the YODA controller simply changes the VIP-to-YODA-instance map-

	Name	Priority	Match	action
1	r-jpg2	3	url=*.jpg	split={D_2=0.5, D_3=0.5}
2	r-css1	3	url=*.css	split={D_1=1}
3	r-css1	2	url=*.css	split={D_3=0.5, D_4=0.5}
4	r-cookie	0	cookie=*	table={cookie-table}

Table 3. YODA Interface.

ping at the L4 LB, and the L4 LB will start forwarding packets based on the new mapping.

This VIP update mechanism raises two new algorithmic challenges due to transient flow migration that can potentially overload YODA instances and/or TCPStore.

Limiting YODA instance load: First, the VIP-to-YODA-instance mapping has to be changed on multiple L4 LB instances, which is not atomic [38]. As a result, during transition a YODA instance may receive some fraction of the traffic based on the new mapping (from the L4 LB instances that got updated), and some on the old mapping (from the L4 LB instances that are yet to be updated). This can potentially overload YODA instances. We address this challenge by repurposing the capacity on the YODA instances reserved for failures to absorb the transient traffic. We add a new constraint where the transient traffic – sum of the max traffic for individual VIPs under the old or the new assignment is under the capacity (Eq. 4,5 in Figure 7).

Limiting TCPStore load: Secondly, VIP assignment change can potentially cause many connections to migrate if the YODA instances that were assigned to one VIP are removed in the next assignment, which can overload the TCPStore. We address this challenge by adding a constraint δ on the number of connections allowed to migrate, determined by the throughput of the TCPStore (Eq. 6,7 in Figure 7). $C_{v,y}^{old}$ denotes the number of connections for the v-th VIP handled by y-th YODA instance, and $m_{v,y}$ denotes whether v-th VIP is removed from y-th YODA instance.

We describe how we collect the inputs to the assignment component in §6, and evaluate the assignment computation time in §8.

5. L7 LB Features

We now detail on how YODA implements L7 LB features.

5.1 Interface

The primary goal of YODA interface design is to enable online service operators to easily express their policies on how to split the traffic. From studying use cases, we find that an OpenFlow-like interface provided by HAProxy is simple yet powerful. HAProxy lets operators declare the *rules* consisting of the equivalent of *match* and *action* fields. In YODA, we reuse the HAProxy interface with the addition of *priority*. Priority helps to reduce the number of rules when expressing load balancing policies (see primary-backup policy below). Below we highlight how this interface can be used to set some of the most common policies.

Weighted-split: In the simplest case, operators can specify the weights on how to split the traffic (rule-1 in Table. 3).

Primary-backup: In many cases, operators deploy services in a primary-backup model, and want to prioritize traffic to a primary server until it fails/overloads. This can be easily achieved using priority functionality in YODA. Operators can specify the same match condition with two actions with different priorities – a higher priority action specifies weights for primary server(s), and a lower priority rule specifies weights for the remaining servers (rule-2,3).

Sticky-sessions: Operators may want traffic from the same user/session to go to the same server handling that session, by matching on HTTP cookies (rule-4).

Least loaded server: In another common case, operators simply want to forward requests to the least loaded server. This can be done by setting the weights to (-1) for all servers.

5.2 Practical issues

We now describe how YODA handles important practical L7 LB issues including configuration changes.

Adapting to user policy change: Online services can change user policies (rules) dynamically during upgrades, new web design, or to add/remove backend servers. YODA's periodic VIP assignment calculation accounts for the number of rules. YODA then simply updates the new rules on the YODA instances where the VIP is assigned. This change does not break existing connections, as YODA instances only apply new policies to new connections. Packets on existing connections continue to be forwarded to their prior assigned server even during soft server removal so that the connections do not break. If the operator removes the server immediately, then it is treated as failure detailed next.

Backend server failure: The YODA monitoring component periodically pings the servers. When a server fails, its connections with YODA instances are terminated[3].

VIP addition and removal: When new services, *i.e.,* VIPs, are added, YODA first runs the VIP assignment algorithm described in §4.4 to calculate the set of YODA instances assigned to the VIP. Next, we add the rules corresponding to the new VIP to the YODA instances where it is assigned. Lastly, we assign the mapping between the VIP and the YODA instances to the L4 LB. The sequence of VIP removal is in reverse of VIP addition.

HTTP 1.1 and HTTP 2.0: In HTTP/1.1, a single TCP connection can be reused for multiple requests, which may match different rules and hence need to be forwarded to different backend servers. YODA supports this by having YODA instances inspect the packets from the clients for the HTTP content. If the server selected for the new request is different than the current one, it closes the old connection

[3] Although not implemented currently, YODA instances can initiate connections with the new server and relay the same request to the new server, transparently to the client.

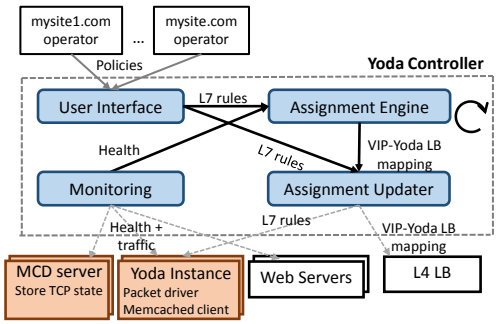


Figure 8. Components in YODA implementation (shown in colored boxes).

and initiates a connection with the new server, and also changes the mapping in TCPStore.

Pipelining requests in HTTP/1.1 poses an additional challenge that YODA instances have to ensure that the responses are sent *in-order*, *i.e.,* in the same order as the received requests, even during YODA failures. YODA achieves this by storing the order in which the requests were received in TCPStore, so that after failure, another instance can gather and forward the responses.

In addition to pipelining requests in HTTP/1.1, HTTP/2.0 and SPDY propose out-of-order delivery of responses to avoid head-of-line blocking [6]. YODA can easily support out-of-order responses from a server by correctly translating the TCP sequence numbers.

SSL support [9]: YODA supports SSL by sending the security certificates (set by the operators) to the clients. On failure during certificate transfer, another YODA instance resends the entire certificate (TCP buffer at the client will remove duplicate packets). The rest of the function remains the same as detailed in §3, except the YODA instance uses the security certificate for SSL termination to decrypt the request to get HTTP content and select the server.

Sending the same request to multiple servers: Many websites send the same client request to multiple backend servers to reduce latency and handle server failures, and send the first server response back to the client. Although it is not currently supported, YODA can easily support this by establishing connections with multiple servers, and tunneling the first response it receives from the servers. It makes a mark to drop packets of later responses from other servers.

6. Implementation

We have implemented YODA on Linux completely at the user-level, *i.e.,* it requires no changes to the existing kernel, and therefore it can be deployed in the public cloud as a service by a third party. We now describe the key components of YODA: (1) YODA instances, (2) TCPStore, and (3) YODA controller, as shown in Figure 8. All components are written in C or python with the total 5k+ lines of code.

YODA **instances:** Each YODA instance runs in a VM and intercepts the packets forwarded from the L4 LB using `nfqueue` and `iptables`[7]. Each instance runs packet driver and Memcached client. The packet driver performs key functions of the YODA instance. It (1) establishes connections with clients/servers by sending and receiving raw packets, (2) tunnels the packets on existing connections by changing the TCP fields using `nfqueue`, (3) stores and retrieves the flow state in/from TCPStore whenever necessary. The packet driver runs in user-space.

Inside a YODA instance, the packet driver creates K queues, one for each of the K cores of the VM, and starts a multi-threaded module to listen on each queue. The packets are forwarded to `nfqueue` using `iptables` such that the packets on the same TCP connection are forwarded to the same queue. This mechanism ensures that the traffic is evenly spread across all the CPU cores, and packets on the same connection are handled in the same order they arrive. We did not use kernel-space SNAT/DNAT options as they do not provide fine-grained control over the TCP fields.

Lastly, each YODA instance keeps track of the traffic for individual VIPs that YODA controller reads periodically.

TCPStore: We run unmodified Memcached on multiple VMs that store the flow state. We modified the Memcached client library to store the same key-value pair on multiple (K) Memcached servers for persistence. For any TCPStore operation, the Memcached client first determines the K servers among the total N servers using K different hash functions, and consistent hashing.

When a Memcached server fails, we do not replicate its key-value pairs mainly because flows finish quicker than the replication latency.

Controller: The controller is at the heart of YODA architecture. It has four components: (1) User interface: It converts the user policies expressed using the YODA interface into the rules and sends them to the YODA instances. (2) Assignment engine: It calculates the assignment between the VIP and YODA instances using the ILP detailed in §4.4 implemented using CPlex [3], (3) Assignment updater: It takes the VIP assignment from the assignment engine and changes the mapping at the L4 LB. (4) Monitor: It gathers health information by pinging the YODA instances, Memcached servers, and backend servers every 600ms, and hence detects failure with at most 600ms delay. It also gets traffic statistics from the YODA instances. All components communicate with each other using RESTful APIs.

7. Evaluation

Our testbed experiments evaluate the key ideas and objectives of YODA design and show that: (1) Decoupling flow state from the L7 LB (or middleboxes in general) and maintaining them in a persistent storage is feasible – it incurs insignificant latency to the flows being balanced and the cost of running the persistent storage is low; (2) YODA provides

high availability and scalability – it does not break flows during LB failures, addition, removal, and user policy updates.

Setup: Our testbed consists of 60 VMs in Windows Azure: 10 act as YODA instances, 10 as Memcached servers, 30 act as the backend servers, and 10 run a version of Ananta L4 LB that forwards packets to remaining YODA instances upon a YODA instance failure. The clients (generating requests) are located on a university campus.

The backend servers are split across 4 online services – each online service emulates a university website storing faculty and student webpages and embedded objects, which are collected from an actual university website. In total we collected 10K+ objects with sizes 1K-442KB (median 46KB). Each web-request fetches an HTML page and its embedded objects[4].

Each YODA and TCPStore instance runs on a separate VM with 8-core CPU and 14GB RAM, and the backend servers run on dual-core VMs with 3.5GB RAM, running the Apache/2.2.3 HTTP server. Each client generates the request workload using either a Python client that emulates web-browser or the `Apache benchmark` tool. All YODA instances, TCPStore servers, backend servers, and clients run Ubuntu 12.04.

7.1 Feasibility of decoupling flow state

We first evaluate the feasibility of decoupling flow state from the YODA instances by measuring its latency overhead to the flows, CPU overhead to the YODA instances, and the scalability of TCPStore.

Latency overhead: To measure the latency overhead, we configure clients to send requests at 50K req/sec for small objects (responses are of size 10KB). Using smaller objects stresses the mechanisms for decoupling flow state since they incur higher load on connection establishment and more frequent operations to TCPStore than larger objects.

To put the latency overhead in perspective, we break down the end-to-end latency (request completion time) into: (1) Baseline: the end-to-end latency when not using any load balancer, which includes the latency incurred by the Internet and backend server processing, (2) Connection: the time an LB instance takes to establish the TCP connection with the backend. (3) Storage: the latency incurred while inserting the flow information into TCPStore, unique to YODA. (4) LB: The remaining latency incurred by the LB instances while processing the packets.

Figure 9 shows the breakdown of median latency for YODA and HAProxy, of all flows in the experiment. We make the following observations. (1) The end-to-end latency in YODA and HAProxy are 151 msec and 144 msec, respectively, out of which the baseline latency is 133 msec. (2) The extra latency to store flow states in TCPStore in YODA is only 0.89 msec. (3) YODA takes slightly longer to establish

[4] We modified the webpages to change the URL for the embedded objects accordingly.

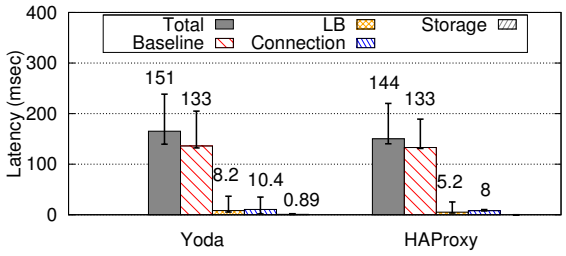


Figure 9. YODA latency breakdown.

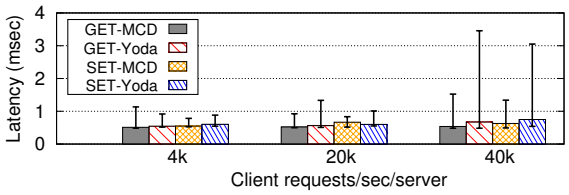


Figure 10. YODA TCPStore latency.

connections – 10.4 msec compared to 8 msec in HAProxy, and to process packets – 8.2 msec compared to 5.23 msec in HAProxy. This is because currently YODA is written in Python and also as it copies packets between user and kernel space while HAProxy performs TCP splicing in the kernel; an in-kernel implementation of YODA will achieve the same latency as HAProxy.

CPU overhead: Next, we measure the CPU utilization on the LB instances. In YODA, the CPU saturates (100% utilization) for 12K client req/sec for small requests[5], and hits 80% for 90K packets/sec request rate for larger requests (flow size 2MB). Under HAProxy, the CPU utilization is 46% and 34% for the two cases. The close to 2x higher CPU utilization in YODA is again due to copying packets between the kernel and user space; we isolated the Memcached client calls and found them to consume negligible CPU load. We fully expect an in-kernel implementation of YODA to reduce the CPU utilization to be similar to that of HAProxy.

TCPStore performance: We first measure the latency overhead of supporting persistence in TCPStore. We measure the latency of `get, set and delete` operations while issuing 40K, 200K, and 400K ops/sec across 10 Memcached servers for 60 seconds. Figure 10 compares the latency for the default Memcached which does not store replicas and the Memcached changed in YODA to provide 2 replicas for persistence. Note the x-axis denotes the number of client requests per Memcached server. Recall for a single client request, a YODA instance issues two `set` operations to store the flow state (§3). We make two observations: (1) even at 40K client req/sec/server, the median operation latency under the default Memcached is only 0.75 msec, which is rather insignificant compared to the median end-to-end la-

[5] [5] noted higher throughput due to high-end physical server.

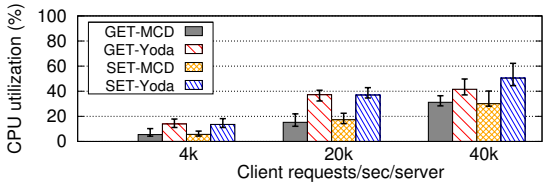


Figure 11. YODA TCPStore CPU utilization.

tency of 151 msec. (2) The overhead of adding persistence to the default Memcached is very small: for 40K req/sec/sever, the overhead for the three operations is less than 24% (0.18 msec). The low latency overhead benefited from sending the operation to two replica servers in parallel.

Figure 11 shows the CPU utilization of Memcached (default and in YODA). As expected, the persistence feature of TCPStore, which issues each operation to two servers, doubles the average CPU utilization.

Our evaluation shows that a single Memcached server can handle 80K client req/sec (at 90% CPU utilization), while each YODA instance can handle 12K client req/sec. This suggests decoupling flow state to support high availability in a scale-out L7 LB design is practical: we just need 1 TCPStore instance deployed for every 6.6 YODA instances.

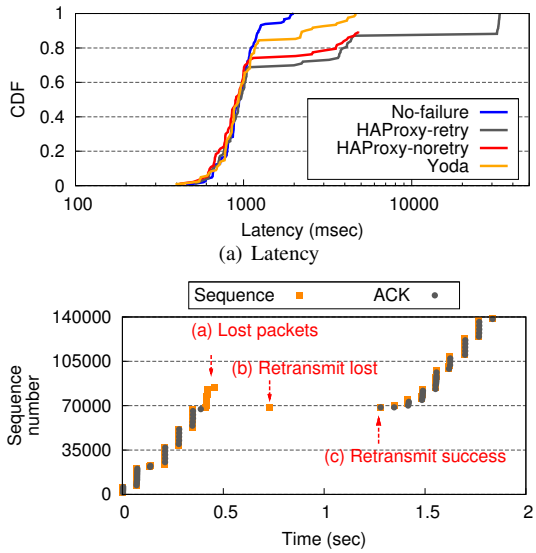
7.2 Failure recovery

One of the most important benefits of YODA is it maintains client flows during YODA instance failures. In this experiment, we start with 10 YODA instances and fail 2 of them simultaneously. Without failure, a webpage and all its objects are downloaded in 840 msec (median case). The clients emulate browser behavior with HTTP timeout set to 30 seconds, which is the least among the popular web browsers we tested (*e.g.,* Android Chrome has 60 sec, and C# HttpWebRequest library has 100 sec [14, 15]).

The clients send requests using 20 processes each. Each process waits for the completion/timeout of the previous request before issuing a new one. We repeat this experiment in four scenarios: (1) HAProxy without browser retry (HAProxy-noretry), (2) HAProxy with browser retry=1 (HAProxy-retry), (3) YODA without browser retry (denoted as YODA-noretry), (4) YODA with browser retry=1 (not shown as there was never any retry made).

Figure 12 shows the CDF of the end-to-end latency for requests. We make three observations: (1) HAProxy-noretry broke 24% of the flows on failure, whereas YODA and HAProxy-retry did not break any flow, (2) YODA increased the end-to-end latency by 0.6 to 3 seconds, and (3) HAProxy-retry increased the latency beyond 30 sec.

To understand how YODA maintained the flow during failure, we plot in Figure 12(b) the `tcpdump` output collected at the backend server for a flow that experienced the YODA instance failure. Upon the failure, (1) the packets from the server and client going through the failed instance are dropped during failover (point a in Figure 12(b)). (2)



(a) Latency

(b) Sequence number for a connection under YODA.

Figure 12. YODA Failure recovery.

The server retransmits the packet first at 300 msec (point b), which the L4 LB sends to the failed LB, as the mapping at the L4 LB is yet to be updated. (4) The server retransmits the packet at 600 msec (point c). At this time, since the YODA monitor has detected the instance failure and updated the mapping at the L4 LB, the packet is sent to one of the online YODA instances. (5) That instance retrieves the flow state from TCPStore and forwards the packet to the client, which sustains the connection with the new YODA instance, and subsequent packets are forwarded normally. Note that the client did not timeout, and no HTTP request was resent.

In contrast, when an HAProxy instance fails, all the packets from the servers and ACKs from the clients going through the failed instance are dropped, breaking the connections with the server and the client. After the HTTP timeout of 30 sec, if the client resends the HTTP request (HAProxy-retry), the request is forwarded to one of the live HAProxy instances as the L4 LB is updated; those objects are successfully retrieved but with a delay of 30 seconds.

7.3 Scalability

In this experiment, we show that YODA scales as the traffic coming to the load balancer increases without breaking existing connections. Figure 13 shows the request rate (req/sec) and CPU utilization during a 30-second long experiment, where we send requests using the `Apache benchmark` tool to fetch individual objects. In this experiment, we initially have 6 YODA instances. At time = 10 sec, we increase the traffic for each YODA instance from 5K req/sec to 10K req/sec, which increased the CPU utilization on the YODA instances from 40% to 80%. As a result, the YODA controller adds 3 more instances to reduce the CPU utilization which reduced the traffic on each YODA instance to 6.7K

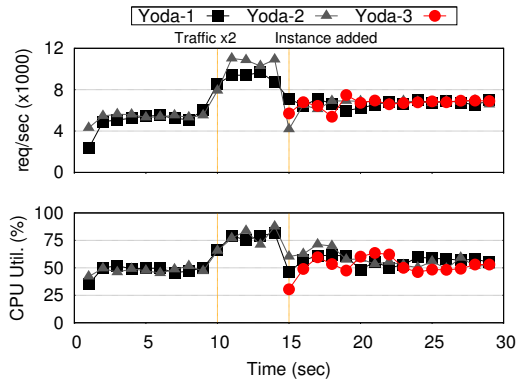


Figure 13. YODA scalability.

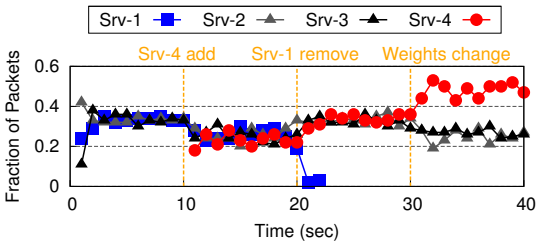


Figure 14. YODA user policy update.

req/sec, and CPU utilization to 60%. Importantly, all client flows were maintained throughout the experiment.

Lastly, we also measure the latency throughout the experiment. Surprisingly, we do not see any significant variation in latency during the dynamics. The latency during the 10-15 second interval under higher traffic load is not inflated because the queues on the YODA instances will not build up until the CPU saturates. The same was observed with the software Mux in Ananta [38].

7.4 Safe policy update

In this experiment, we show that YODA can safely update user policies without breaking existing flows. We also show that the weights set to split the traffic in user policies are correctly implemented by the distributed YODA instances.

In this experiment, initially the user policy specifies equal weights among the 3 backend servers running on identical VMs. The operator wants to replace one of the VMs with a VM that has 2x CPU cores. To do this, the operator uses make-before-break, where at 10 sec, the operator adds the new VM (Srv-4), and at 20 sec, the operator removes one of the existing VMs (Srv-1). Lastly, at 30 sec, the operator changes the weights among the servers (Srv-2,3,4) to 1:1:2.

Figure 14 shows the dynamics during the 40-second interval. We see that the fraction of traffic and CPU utilization (not shown) change according to the policy changes. Between 0-10 sec, server -1 to -3 receive equal traffic. As the new VM is added at 10 sec, the traffic is now split equally across the 4 servers. At 20 sec, as Srv-1 is removed, the traffic is split equally among the 3 servers. Finally, at 30 sec, the traffic is split following the 1:1:2 ratio. Again, none of the client flows were broken as YODA adapts to the user policy changes (§5.2).

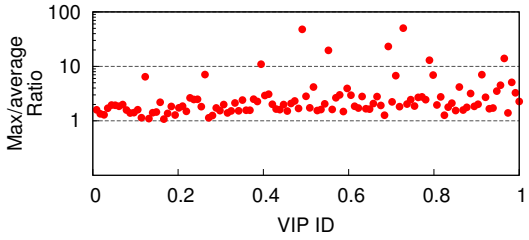


Figure 15. Max-to-average traffic ratio for all VIPs. VIPs are sorted in decreasing order of the traffic.

8. Simulation

In this section, we evaluate YODA's VIP assignment algorithm (§4.4) using a traffic trace production cloud. We show that: (1) YODA reduces the L7 LB cost by 3.7x on average when all online services share the LB. (2) YODA update algorithm is effective in limiting the transient traffic overload and flows migrated;

Setup: The traffic trace collected from our production cloud consists of all flows received by the Internet-facing services in a 24-hour period (during a weekday). The trace consists of 100+ VIPs and 50K+ L7 rules. We calculate the assignment between the VIP and the YODA-instances every 10 mins. The VIP assignment algorithm finishes in 1.5-21.5 sec (median 3.92 sec) using the CPLEX ILP solver.

8.1 Cost reduction

We estimate the cost reduction based on the ratio of the max. to average traffic for the individual VIPs, as the max-to-average ratio indicates the cost savings possible by using the elasticity of YODA-as-a-Service. This is because in using HAProxy, each individual online service would need to provision LB based on its peak traffic demand, as dynamically removing/adding HAProxy instances could break the connections, whereas in using YODA-as-a-service, an online service can easily scale up/down its share of LB instances without breaking connections, and just has to pay for its share of YODA instance usage, and over time, its average LB instance usage is proportional to its average traffic load. Note we are ignoring discretization effect in both cases, which makes the saving conservative.

We calculate the ratio of the max-to-average traffic for individual VIPs in every 10-min interval throughout 24 hours. Figure 15 shows the max-to-average ratio for all the VIPs. The VIPs are sorted in decreasing order based on their traffic volume. By using YODA, these online services can save

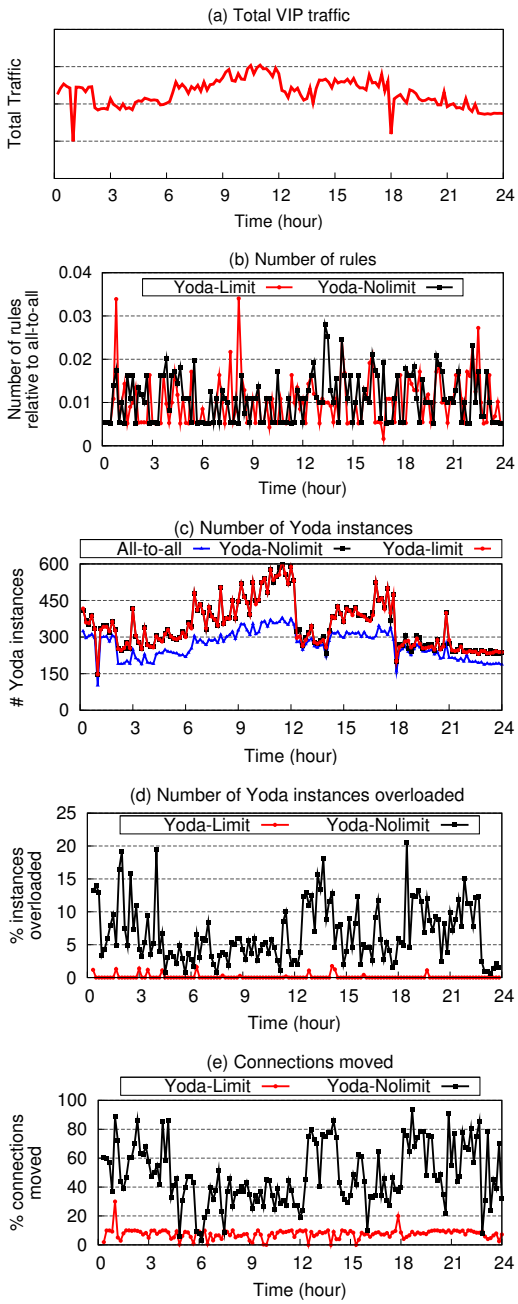


Figure 16. YODA update.

L7 LB cost by 1.07x to 50.3x. (average = 3.7x across all VIPs).

Next, we show that YODA can provide these cost benefits by frequently calculating (every 10 mins) the VIP to YODA instance mapping (§4.4), and updating it through congestion-free update (§4.5).

8.2 Impact of updates

We evaluate the effectiveness of YODA's assignment algorithm in terms of (1) the number of YODA instances required, (2) fraction of the YODA instances overloaded during transition, (3) number of flows migrated. The smaller these values, the more effective the assignment algorithm.

We set the target latency due to YODA to 5 msec, which translates into 2K rules on each YODA instance (Ry=2K) based on Figure 6. We set the limit on the number of flows to be migrated to 10%. Lastly, we set $n_v = 4 \cdot \frac{t_v}{T_y}$, *i.e.,* each VIP gets 4x more replicas by using YODA as a shared service than using YODA individually.

We compared two versions of the algorithm: (1) YODA-no-limit where there is no limit on the transient traffic or number of flows migrated, (2) YODA-limit, which provides congestion-free transition, and may require more instances.

Numbers of rules: Figure 16(b) shows the median number of rules across the LB instances that YODA-no-limit and YODA-limit generate normalized to that by the all-to-all scheme. We see that YODA instances store 0.5-3.7% (median 1%) of the rules compared to Ananta, which reduces the number of rules in YODA by 100x compared to Ananta. But this also comes at a cost of increasing number of instances.

Number of instances: Figure 16(c) shows the number of YODA instances for YODA-limit and YODA-no-limit. As a reference, we also show the result for a base-line all-to-all assignment, which requires least number of instances *i.e.,* the total traffic divided by traffic capacity of each instance.

We make two observations: (1) YODA-no-limit (and -limit) requires 4.6-73% (average = 27%) more instances than the all-to-all. This is the overhead of reducing the number of rules. (2) Importantly, YODA-limit only requires up to (-8) – 11.7% more instances compared to YODA-no-limit (median 1.3%). In some cases YODA-limit required lesser instances because optimality gap in CPLEX was set to 10%. This shows that YODA assignment algorithm is effective in *packing* VIPs under the constraints and the overhead of providing congestion-free transition is very small.

Transient overload: Next, Figure 16(d) shows over the 24-hour period, YODA-no-limit results in 0-20.4% (median 5.3%) of the YODA instances overloaded during transition, which is significantly reduced in YODA-limit. The instances that were overloaded in YODA-limit were already overloaded before starting the new round. This also emphasize the need to frequently update the VIP assignment.

These result confirm that YODA-no-limit results in significant number of YODA instances overloaded during transition compared to YODA-limit.

Number of flows migrated: Figure 16(e) shows that under YODA-no-limit 2.7-95% (median 44.9%) flows migrated from one YODA instance to other, but under YODA-limit the number is significantly lower at 0-29.8% (median 8.3%). We set the limit on the number of flow migration to 10%, but the LP gave infeasible assignment at two points. In such cases, we increased the limit by increments of 10%, and the LP gave a feasible assignment when the limit was 30%.

9. Questions and Answers

We discuss a few questions that are frequently asked.

Q: Is it necessary to separate L4 and L7 LB?
A: We propose a separate L7 LB service for two reasons: (1) A large fraction of the traffic in datacenters do not need to go through L7 LB but need L4 LB. (2) Building the L7 LB service on top of L4 LB follows the module design principle.

Q: Why are there thousands of rules per tenant?
A: The rules are specified on the HTTP, TCP and IP fields. There could be a large number of rules, as there are billions of URLs and cookies.

Q: Does YODA instance access TCPStore for every packet?
A: No. TCPStore is accessed while establishing connection and after YODA instance failures.

Q: Does an L4 instance failure affect YODA's reliability?
A: No. L4 LB has built-in resilience to instance failures [38].

Q: Can the L4 LB failure resiliency mechanism used in L7?
A: No. L4 LB selects the server by hashing TCP/IP tuples, which are carried by every packet of a flow. In L7 LB, the HTTP header used for server selection is not embedded in every packet.

10. Related Work

To our knowledge, YODA provides the first highly available cloud-scale L7 LB design. In the following, we review related work on other middlebox functions in the cloud and network state management and migration.

Middleboxes in cloud: Several startups provide middlebox functions in the cloud. (*e.g.,* Aryaka [10], Barracuda [2] Qualys [13]). APLOMP [42] helps online operators split middlebox functionality between the cloud and enterprise, which motivates the need for 3^{rd} party offered middlebox functionalities in the cloud. Additionally, FTMB [43] and [40] show the evidence of middlebox failures in cloud which facilitates the need for highly available middleboxes.

Layer-4 LB: There are several designs proposed for scalable layer-4 LB in cloud. Ananta [38] uses software instances, whereas Duet [20] and Rubik [21] use hardware switches and software instances for load balancing. The principles used by layer-4 LB designs in providing availability cannot be used in layer-7 LB. Specifically, the layer-4 LBs use IP 5-tuple to select the server, and all the packets on a given connection carry the same IP tuple. Therefore, even if the packets on the same connection reach different LB instances, the LB instances select the same backend server. On the other hand, the HTTP header is only in the first few packets, which necessitates the need to store the selected backend persistently.

State management: OpenNF [22] helps to scale middleboxes safely by moving the flows and state from one middlebox to another. However, its focus is on providing elasticity and not failure resiliency, and thus it does not explicitly decouple flow state and store them in a persistent storage for transparent flow recovery in case of failure. FTMB [43] has high cost of maintaining the state. SSM [29] maintains user sessions, which YODA can leverage to maintain user sessions in addition to connections. Prior work has looked at seamless migration of the BGP sessions and router failures [27, 28], which is different goal that YODA.

Request redirection and CDN: DONAR [49] and Oasis [19] focus on request redirection across multiple DCs and enterprises through DNS. Centrifuge [16] focuses on lease management of accesses to data that are partitioned across in-memory servers. It does not recover state lost in failure and assumes applications will recreate the lost state.

TCP splicing, handoff and migration: Application layer proxies typically use TCP splicing [34, 46] to bind the sockets sockets connected to the server and client. But these designs break the flows when proxies fail. Work on TCP handoff [17, 37] focus on bypassing the LB instance (front-end) on the reverse path and do not address LB instance failure. TCP Migration also tries to maintain connections when servers fail or in the presence of IP mobility. But they require changes at the clients transport layer [44, 45] or socket API [33] or assigning the same IP to all servers [35, 47] which cannot work in the cloud.

Rule assignment and packet processing: Although rule management is not a main focus, YODA can benefit from the recent work on rule management and fault-resilient updates [23, 26, 30, 31, 36, 50]. Additionally, YODA can leverage recent works to improve its packet processing performance within individual instances [39, 51].

11. Conclusion

YODA is a distributed L7-LB-as-a-service designed to meet the availability, scale and operational requirements of multi-tenant clouds. The high availability in YODA is attributed to three design choices: (1) decoupling the TCP state from individual instances and storing it in a persistent in-memory storage (called TCPStore). (2) A novel mechanism that enables one instance to re-use TCP state created on different instance. (3) Using virtual IP to establish connections with the clients and servers so that the connections are shielded from the failure and other dynamics within the YODA instances (front-and-back indirection). Our evaluation of YODA using a prototype implementation and simulations using a production traces shows that YODA can transparently restore flows with low latency inflation ($<$ 1msec) while meeting high availability and scalability requirements. Additionally, compared to used for individual tenants, YODA as a shared service can reduce the L7 load balancing cost by 3.7x.

Acknowledgment

We would like to thank the anonymous reviewers and our shepherd Anja Feldmann for their valuable feedback.

References

[1] A10 networks ax series. http://www.a10networks.com.

[2] Barracuda lb. https://techlib.barracuda.com/display/blbv42/3539006.

[3] Ibm cplex lp solver. http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/.

[4] F5 load balancer. http://www.f5.com.

[5] Ha proxy load balancer. http://haproxy.1wt.eu.

[6] Http 2.0. http://https://tools.ietf.org/html/draft-ietf-httpbis-http2-04.

[7] Linux iptables and nfqueue. http://www.iptables.info/en/iptables-targets-and-jumps.html.

[8] Memcached key-value store. memcached.org.

[9] Ssl. http://en.wikipedia.org/wiki/Transport_Layer_Security.

[10] Aryaka networks, . https://http://www.aryaka.com/.

[11] Amazon web services, . http://aws.amazon.com.

[12] Google cloud platform, . http://cloud.google.com.

[13] Qualys inc, . https://www.qualys.com.

[14] Firefox http timeout, . https://support.mozilla.org/en-US/questions/998088.

[15] Httpwebrequest timeout, . http://msdn.microsoft.com/en-us/library/system.net.httpwebrequest.timeout.

[16] A. Adya, J. Dunagan, and A. Wolman. Centrifuge: Integrated lease management and partitioning for cloud services. In *Proc. of USENIX NSDI*, 2010.

[17] M. Aron, P. Druschel, and W. Zwaenepoel. Efficient support for p-http in cluster-based web servers. In *Proc. of USENIX ATC*, 1999.

[18] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *Proc. of USENIX NSDI*, 2016.

[19] M. J. Freedman, K. Lakshminarayanan, and D. Mazières. Oasis: Anycast for any service. In *Proc. of USENIX NSDI*, 2006.

[20] R. Gandhi, H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. In *Proc. of ACM SIGCOMM*, 2014.

[21] R. Gandhi, H. H. Liu, Y. C. Hu, C.-K. Koh, and M. Zhang. Rubik: unlocking the power of locality and end-point flexibility in cloud scale load balancing. In *Proc. of USENIX ATC*, 2015.

[22] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. Opennf: Enabling innovation in network function control. In *Proc. of ACM SIGCOMM*, 2014.

[23] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource fair queueing for packet processing. In *Proc. of ACM SIGCOMM*, 2012.

[24] J. Hamilton. The cost of latency. http://perspectives.mvdirona.com/2009/10/31/TheCostOfLatency.aspx.

[25] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford. Niagara: Scalable load balancing on commodity switches. In *Technical Report (TR-973-14), Princeton*, 2014.

[26] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford. Efficient traffic splitting on commodity switches. In *Proc. of ACM CoNEXT*, 2015.

[27] E. Keller, M. Yu, M. Caesar, and J. Rexford. Virtually eliminating router bugs. In *Proc. of ACM CoNEXT*, 2009.

[28] E. Keller, J. Rexford, and J. Van Der Merwe. Seamless bgp migration with router grafting. In *Proc. of USENIX NSDI*, 2010.

[29] B. C. Ling, E. Kiciman, and A. Fox. Session state: Beyond soft state. In *Proc. of USENIX NSDI*, 2004.

[30] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. zupdate: Updating data center networks with zero loss. In *Proc. of ACM SIGCOMM*, 2013.

[31] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter. Traffic engineering with forward fault correction. In *Proc. of ACM SIGCOMM*, 2014.

[32] H. Lu, J. Yang, Z. Liu, N. D. Lane, T. Choudhury, and A. T. Campbell. The jigsaw continuous sensing engine for mobile phone applications. In *Proc. of the 8th ACM Conference on Embedded Networked Sensor Systems*, 2010.

[33] D. Maltz and P. Bhagwat. Msocks: An architecture for transport layer mobility. In *Proc. of IEEE INFOCOM*, 1998.

[34] D. A. Maltz and P. Bhagwat. Tcp splice application layer proxy performance. *Proc. of J. High Speed Netw.*, 2000.

[35] M. Marwah, S. Mishra, and C. Fetzer. Fault-tolerant and scalable tcp splice and web server architecture. In *Proc. of IEEE SRDS*, 2006.

[36] M. Moshref, M. Yu, A. Sharma, and R. Govindan. Scalable rule management for data centers. In *Proc. of USENIX NSDI*, 2013.

[37] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proc. of ACM ASPLOS*, 1998.

[38] P. Patel et al. Ananta: Cloud scale load balancing. In *Proc. of ACM SIGCOMM*, 2013.

[39] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of open vswitch. In *Proc. of USENIX NSDI*, 2015.

[40] R. Potharaju and N. Jain. Demystifying the dark side of the middle: A field study of middlebox failures in datacenters. In *Proc. of ACM IMC*, 2013.

[41] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan. Timecard: Controlling User-Perceieved Delays in Server-based Mobile Applications. In *Proc. of ACM SOSP*, 2013.

[42] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's

problem: Network processing as a cloud service. In *Proc. of ACM SIGCOMM*, 2012.

[43] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, et al. Rollback-recovery for middleboxes. In *Proc. of ACM SIGCOMM*, 2015.

[44] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proc. of ACM Mobicom*, 2000.

[45] A. C. Snoeren, D. Anderson, and H. Balakrishnan. Fine-grained failover using connection migration. In *Proc. of Usenix Symposium on Internet Technologies and Systems*, 2001.

[46] O. Spatscheck, J. S. Hansen, J. H. Hartman, and L. L. Peterson. Optimizing tcp forwarder performance. *Proc. of IEEE/ACM Trans. Netw.*, 2000.

[47] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory tcp: Highly available internet services using connection migration. In *Proc. of ICDCS*, 2002.

[48] R. Wang, D. Butnariu, and J. Rexford. Openflow-based server load balancing gone wild. In *Proc. of USENIX HotICE*, 2011.

[49] P. Wendell, J. W. Jiang, M. J. Freedman, and J. Rexford. Donar: Decentralized server selection for cloud services. In *Proc. of ACM SIGCOMM*, 2010.

[50] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with difane. In *Proc. of ACM SIGCOMM*, 2010.

[51] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proc. of ACM CoNext*, 2013.