TNSM-2023-06297

Yuz: Improving Performance of Cluster-Based Services by Near-L4 Session-Persistent Load Balancing

Mohammad Hosseini, Sina Darabi, Amir Hossein Jahangir, Ali Movaghar

***Abstract*— Large-scale services are deployed in data centers using clusters of servers, and load balancers (LB) are responsible for distributing requests for a service among its servers. Layer-4 (L4) LBs process requests faster than Layer-7 (L7) ones, but they cannot provide session-persistent load balancing, and therefore, they direct connections of an application-level session to different servers. On the other side, L7 LBs can direct all requests of an application-level session to the same server, but they have a very limited capacity because they act as a reverse-proxy and process requests at the application layer. We present "Yuz", a stateless session-persistent load balancer that does not act as a reverse-proxy. Yuz works near layer 4, and it makes use of TLS session data instead of processing incoming requests at the application level. Our evaluations show that the request rate that can be handled by cluster-based services equipped with Yuz is twice as high as when the clusters use the best existing load balancers. Yuz also significantly reduces the average and tail of the clusters' response time. Moreover, while each of the existing session-persistent LBs works only for a specific application, Yuz provides an application-independent session-persistent load balancing.**

***Index Terms*—Clustered service, Load balancer, Session-persistent, TLS.**

I. INTRODUCTION

IN the late 1990s, the rapid rise of demand for Internet-based services led to the emergence of data centers, where high-demand Internet-based services such as websites are replicated on multiple servers to increase the capacity of processing incoming requests. Data centers rely on the key role of load balancers (LB) to distribute the requests among the servers and cope with the huge amount of traffic load. The design and method of a load balancer have a significant impact on the performance of services running on such clustered servers. Hence, many studies have been carried out to improve the design of LBs. Basically, the load balancing approaches can be classified into two categories according to the OSI protocol stack layer at which they operate: Layer-4 (L4) and Layer-7 (L7) load balancing.

Layer-4 load balancers are typically simpler and faster than layer-7 ones because they only deal with network-level connection information such as IP addresses and port numbers. The main issue in the context of L4 load balancing is *connection affinity* (also known as *connection persistence*), that is, to direct packets belonging to the same connection to the same server. When a layer-4 LB receives a new connection request packet (TCP SYN) from a client, it selects one of the servers based on a load balancing logic and then forwards the packet and all the subsequent packets that have the same 5-tuple to the selected server until the connection is closed. The dynamic nature of servers and load balancers in data centers makes the connection affinity issue an important problem addressed by several studies [1]–[5].

Another important issue regarding L4 load balancers is that they cannot recognize application-level sessions. A session is a series of transactions issued by the same client during an entire work and interaction with an application, and it can store temporary information related to the activities of the client. For example, in web applications, this can include filling out a series of forms on a website, adding items to a shopping cart, or checking a mailbox and sending emails after logging in. The transactions can result in many network-level connections (TCP) established and closed during the time the session is open. A session can last from a few minutes to several hours and even days [6]. The important point is that sessions cannot be recognized by network-level information such as IP addresses and port numbers. Therefore, applications deal with their session identifiers in the application layer, which is invisible to L4 load balancers.

A consequence of session-unaware load balancing is that TCP connections of an application session are directed to different servers during the time the session is open. As discussed in the next section, it imposes significant performance penalties on the services that keep and use sessions. On the other side, layer-7 load balancers [7]–[10] can provide *session persistence* (also known as *client affinity*). Being session-persistent means that when an application-level session is established between a client and a server, the load balancer directs all subsequent TCP connections of the client's session to the same server until the session is closed. L7 load balancers act as a reverse-proxy; that is, they terminate each incoming connection request, process application-layer information of the request, and make a new connection to the server. Hence, acting as a reverse-proxy makes the capacity of L7 load balancers very limited [11]. Ideally, we need a load balancer that can process incoming requests as fast as L4 load balancers while also providing session persistence.

In this paper, we present "**Yuz**", a high-performance session-persistent load balancer that does not act as a reverse-proxy. This paper is the extended version of our short paper

Mohammad Hosseini is with the Faculty of Computer Science and Engineering, Shahid Beheshti University, Tehran, Iran. (e-mail: corresponding author: m-hosseini@sbu.ac.ir)

Sina Darabi is with the School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran 19538, Iran. (e-mail: s.darabi@ipm.ir).

Amir Hossein Jahangir and Ali Movaghar are with the Computer Engineering Department, Sharif University of Technology, Tehran 14588, Iran (e-mail: jahangir@shairf.ir; movaghar@sharif.ir).

presented in CNSM`21 [12]. In the previous short paper, we briefly discussed how the session data and the session resumption mechanism of Transport Layer Security (TLS) 1.2 protocol can be utilized to achieve an application-level session-persistent load balancer for clustered web servers without processing information in the application layer or even terminating TLS sessions. In this paper, we discuss the approach in detail and present the fully matured method, which has become a completely stateless solution. The method works with not only TLS 1.2 but also TLS 1.3, which is the most recent version of the protocol. Unlike existing session-persistent load balancers, each of which works for only a specific application such as Web, the proposed method works regardless of the application type running on the servers. The only necessary condition is that the application is required to use TLS, the standard and widespread protocol used by Internet-based applications to secure their connections. We have carried out a realistic evaluation by measuring the performance of a cluster-based service equipped with different load balancers. The evaluation shows that the cluster-based service can handle a significantly higher request rate when it is equipped with Yuz. Moreover, Yuz significantly decreases the response time of the clustered system. Our contributions can be summarized as follows:

- We demonstrate how session-persistent load balancing can improve the performance of cluster-based services.
- We investigate the limitations of existing L4 and L7 load balancers and present an experiment that reveals how the high processing cost of L7 LBs and the session-unaware load balancing of L4 LBs limit the performance of cluster-based services.
- We present Yuz, the first session-persistent load balancer whose processing cost is near existing L4 load balancers. Moreover, Yuz is the first session-persistent LB that is fully stateless, which means it can be easily and efficiently deployed in a scalable manner. Furthermore, thanks to our new connection hand-off mechanism, Yuz is the only session-persistent LB that is not application-specific, which makes it suitable for cloud environments.
- We conduct an evaluation to assess the performance of a cluster-based service provided with state-of-the-art L4 and L7 load balancers. Our evaluation is the first evaluation in the literature that takes application sessions into account, which leads to a more realistic evaluation. It shows that when the cluster is equipped with Yuz, the cluster can handle a request rate about twice as high as when it is provided with the best existing solution. Moreover, the evaluation shows that Yuz reduces the average and tail of the cluster's response time by a factor of 2 and 14, respectively.

The rest of this paper is organized as follows: in Section II, after reviewing the load balancing background, we illustrate the problem of session-persistent load balancing. Section III presents the proposed method, and in Section IV, it is evaluated. In Section V, we discuss some important issues regarding the proposed method. Finally, Section VI concludes the paper and presents future works.

II. Background and Motivation

In this section, we outline the architecture of a cluster-based service, provide a brief background of load balancers, and then discuss the importance of session-persistent load balancing and the limitations of existing load balancers regarding this issue. Before discussing the background of load balancing, it is important to note that there are two types of load balancing in data centers, which are orthogonal to each other: flow (or network) load balancing, and server load balancing. The first, which is related to the routing problem, focuses on the network and tries to minimize the congestion on network links and consequently reduce the flow completion time tail latency by distributing the flows evenly among the internal paths and links of a data center [13]–[18]. The second focuses on the servers and aims at distributing the load of requests evenly among them and consequently increasing the request processing capacity. We show that our server load balancing scheme not only increases the capacity but also can lead to a significant reduction in the average and tail of end-to-end request completion time, i.e., the time it takes between sending the request and receiving its response.

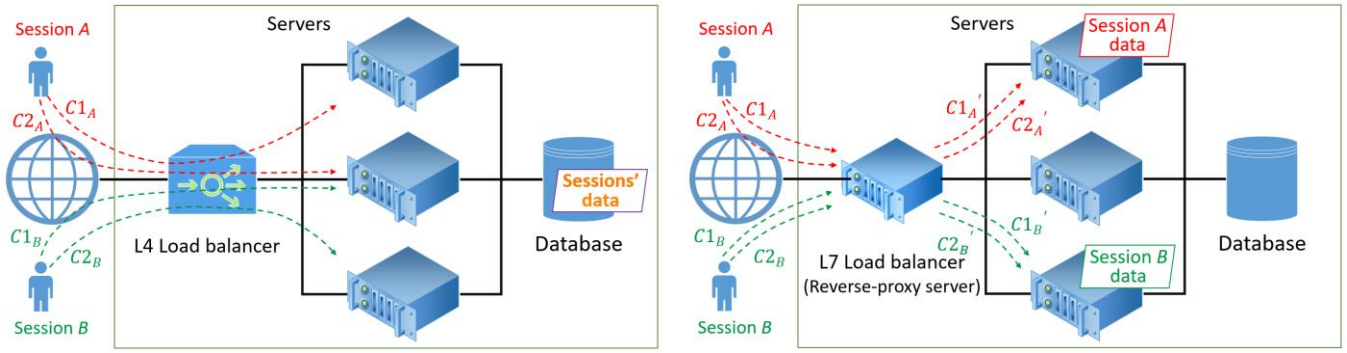
A. Background of L4 Load Balancers

A cluster-based system consists of a collection of servers tied together in a data center to act as a single entity providing a service. Each server of the service is assigned a private IP address in the data center network, which is called *Direct IP* (DIP). The service itself has a public IP address called *Virtual IP* (VIP) by which it is identified on the Internet. A load balancer, as the front-end node of the service, receives incoming requests destined for the service (VIP) and evenly distributes them among the servers (DIPs).

Early load balancers were implemented as dedicated hardware devices. As mentioned in [1], [19], the traditional hardware load balancers were not scalable, and it did not take long for them to fall short of the capacity needed for the rapid growth of network traffic. Hence, scalable load balancing systems were introduced which deploy a dynamic cluster of load balancers in front of the service's cluster of servers [1]–[4], [19]. The scalable load balancers typically run on commodity servers or programmable switches of data centers. The border router of data centers splits the traffic between LBs using equal-cost multipath routing (ECMP).

An important problem of scalable load balancers is to provide connection affinity, especially when the set of LBs or back-end servers changes due to failure, upgrading, or adding/removing servers to handle different traffic loads. The approaches to this problem can be divided into two categories depending on whether they store per-connection state or not, i.e., *stateful* or *stateless*.

Ananta [1] and Maglev [2] are well-known stateful software load balancers proposed in the literature for data centers. They keep a connection-to-server mapping table in



(a) Layer-4 load balancing (session-unaware)

(b) Layer-7 load balancing (session-persistent)

Fig. 1. Clustered servers equipped with layer-4 (L4) and layer-7 (L7) load balancers. (There are two client sessions: session A and session B. Dashed lines represent connections. Each session has established two connections at different times, for example, connection $C1_A$ during time t_1 to t_2, and connection $C2_A$ during t_3 to t_4, where $t_3 > t_2$)

each load balancer machine. Each load balancer machine forwards packets to servers by looking up the 5-tuple of packets in its internal connection table. However, Ananta has failed to address the connection affinity issue when the pool of LBs changes. As a result of changing the number of load balancers, the mapping of ECMP changes, and the border router directs packets of ongoing connections to load balancers that do not have any state for them. As for Maglev, it does not guarantee connection affinity when the pool of servers changes because its load-balancing logic is carried out by consistent hashing. Niagara [3] addresses this problem by making use of a centralized Open-Flow-based controller. The controller can share the connection states among load balancers. Therefore, if a packet is directed to a wrong load balancer by ECMP, the controller can give the load balancer the identifier of the server that is in charge of the packet's connection. However, the controller not only adds latency overhead but also negatively affects forwarding performance.

Stateful approaches, which are based on storing a connection table, suffer from important disadvantages. The considerable time of 5-tuple insertion and lookup operations in the table negatively affects their performance. Moreover, the space limitation of the table makes the stateful methods prone to SYN flood attacks. To cope with these problems, stateless load balancers were proposed. Beamer [4] is a stateless LB. It maps and forwards packets to servers using an improved version of consistent hashing [20]. Nevertheless, since consistent hashing does not guarantee connection affinity, Beamer uses a mechanism named "daisy chaining" to deal with wrong connection-to-server mappings. In this mechanism, if a server receives a packet for which it does not have a state, the server tries to forward it to the server that is responsible for the packet's connection. The state-of-the-art load balancing method, Cheetah [5], [21], has adopted a stateless approach as well. It decouples the load balancing logic from the connection affinity mechanism and consequently can easily achieve a uniform distribution of connections among servers. To deal with the connection affinity problem, Cheetah encodes the identifier of the selected server into the TCP timestamp options field of packets directed toward the client. On the other side, the client is expected to include the identifier in the subsequent packets that it sends to the server. Hence, Cheetah can process packets with less processing cost than the previous methods, and it has achieved the highest capacity compared to the other load balancers. However, there is no guarantee that all clients echo the TCP timestamp options back to the servers; most notably Microsoft Windows does not echo the option field as mentioned in the Cheetah paper.

Some other load balancing research, namely Duet [19], SilkRoad [22], and Tiara [23], have focused on implementing L4 load balancers on programmable data planes using P4. It allows them to achieve line-rate throughput on several high-bandwidth links. It is noteworthy that the most recent L4 load balancing methods, Beamer and Cheetah, have achieved line-rate throughput (over one 40G and one 100G link, respectively) in their software implementation as well.

B. The Issue of Sessions and the L7 load balancers

All the mentioned studies have focused on scalable L4 load balancing. Their goal is to provide a better L4 load balancing system in terms of scalability, availability, connection affinity, and uniform distribution. However, none of the L4 load balancers can provide session persistence, a feature that is currently offered by L7 load balancers, such as Nginx[1] [9], HAProxy [10], and Yoda [7] for Web applications. We demonstrate the importance of sessions and session-persistent load balancing by the simple example illustrated in Fig. 1.

As stated before, each application session can result in several connections established and closed during the time the session is open. L4 load balancers cannot recognize sessions, and consequently, they direct different connections of a session to different servers. Fig. 1a depicts an example of this, where connections $C1_A$ and $C2_A$ of session *A* are directed to two different servers. Hence, while the session is open, clients' session data (also known as session context), such as cart items in an online shopping application, must be saved in a shared database so that all the servers that deal with the

[1] The feature can be enabled using the sticky directive in the configuration file of Nginx machines.

session can access the session data. When a server receives a connection request, it processes the request at the application layer and determines the request's session. Then, the server gets the session data from the shared database (in case of a request that is related to an open session), processes the request, and saves all modifications to the session data in the database to be used by the next connections of the session, which may be served by other servers[2]. The back-and-forth transfer of session data between the servers and database, along with consistency problems (mutual exclusion), significantly affects the performance of this approach. At heavy loads, the queuing time of requests for getting the session data can increase the mean, and especially, the tail of response time. The long tail latency directly leads to bad user experiences and loss of revenue in interactive services and web applications that must respond quickly [18], [24].

On the other side, L7 load balancers, which consider and analyze application-level information of incoming requests, can recognize sessions and perform session-persistent load balancing. As a result of having a session-persistent load balancer, all connections of a session are directed to the same server. Therefore, while a session is open, the server that is in charge of the session can save the session data in its internal memory rather than a shared database (as depicted in Fig. 1b). It means the server can process requests of sessions faster and with less latency because it does not need to read/write session data from/to a shared database for each new connection of an open session. Moreover, when it comes to session-persistent load balancing, consistency problems become off-topic issues.

However, the session persistence ability of existing load balancers does not come for free. Current L7 load balancers act as a *reverse-proxy*. A reverse-proxy terminates each incoming connection request (for example, connection $C1_A$ in Fig. 1b) and processes application layer information such as HTTP cookies to identify the request's session. After looking up the server that is in charge of the session, the load balancer makes a new connection ($C1_A'$) to the server and relays data between the client and the server. Taking a web application as an example, a layer-7 LB becomes a complete web server when it acts as a reverse-proxy. Algorithm 1 shows the details of such a reverse-proxy.

Compared to the L4 load balancing approach, this procedure imposes a heavy processing cost and significantly limits the capacity of L7 load balancer machines. In other words, a layer-7 load balancer machine can handle a considerably lower request rate and traffic than layer-4 ones. It means that L7 load balancers are more prone to becoming a bottleneck and limiting the performance of clustered services than L4 load balancers. Our experiment, which is presented in Section IV, confirms it. Prism [8], which is an improved reverse-proxy, has alleviated the problem by handing off TCP

Algorithm 1. The procedure of a layer-7 load balancer (reverse-proxy)

```
Input: msg, client (a message from a client)
1:  if (msg is NEW_TCP_CONNECTION) then
        // Terminate TCP connection request
2:      tcp_client_socket ← TCPAccept(client);
3:  else if (msg is NEW_TLS_CONNECTION) then
        // Perform TLS handshake with client
4:      tls_client_socket ← TLSHandshake(tcp_client_socket);
5:  else
        // Receive and decrypt client's request using TLS
6:      request ← TCPReceiveAndDecrypt(msg,
                                       tls_client_socket);
        // Extract application-level session ID from the request
        // and lookup the server responsible for it
7:      session_id ← GetSessionID(request);
8:      server_id ← LookUpServer(session_id);
        // Send the request to the corresponding server
9:      tcp_server_socket ← TCPNewConnection(server_id);
10:     TCPSend(tcp_server_socket, request);
        // Receive the response from the server
        // and encrypt and send it using TLS
11:     response ← TCPReceive(tcp_server_socket);
12:     TLSEncryptAndSend(response, tls_client_socket);
13: end if
```

connections to the back-end server. This solution resolves the problem of the high processing cost of relaying traffic between clients and servers. However, it does not address the problem of processing requests and determining sessions at the application layer by the load balancer. We discuss Prism's approach in more detail in the next section, and we also show why its hand-off mechanism is not efficient. Furthermore, the stateful design of L7 load balancers imposes scalability problems. They maintain the session-to-server mapping in either a local memory or a shared database. Again, the former (which is adopted by Nginx[3]) can fail when the set of load balancers changes, and the latter (which is adopted by Yoda [7] to become fault tolerant) imposes latency and bottleneck.

Hence, the clustered server systems that deal with millions of requests per second adopt L4 load balancing (such as the Maglev solution [2], which is used by Google) and handle clients' intermediate information using a shared database. Therefore, we aimed to enrich existing L4 load balancers with the session persistence feature. We ended up with a high-performance near-L4 load balancing system that not only provides a traffic processing capacity and throughput as high as L4 load balancers but also features session-persistent load balancing and reduces the tail latency of responses.

It is noteworthy that the session persistence is not the only advantage of L7 load balancers or reverse-proxies, and our solution does not replace them. For example, they can direct incoming requests to appropriate servers according to the requested content. Sometimes, L7 load balancers are placed after L4 load balancers to benefit from L7 load balancing features. However, when there are several load balancers in each tier of L4 and L7 load balancers, the L7 load balancers

[2] To implement this scheme in PHP, session handlers such as on_session_read() and on_session_write() are overridden to utilize a shared database as the storage for session data, and the custom handlers are registered using session_set_save_handler() function. ASP.NET provides this functionality internally, and it can be enabled by setting sessionState mode to "SQLServer" or "StateServer" (instead of "InProc") in the Web.config file.

[3] Nginx works around the problem by defining a shared zone where load balancer machines periodically exchange their session states. It is defined by the zone_sync directive in the configuration file of each Nginx machine.

cannot provide the session persistence feature because connections of a session are directed to different L7 load balancers by L4 ones. We propose Yuz for those who need a high-performance load balancer featuring session persistence. In other words, we add the session persistence feature to L4 load balancers. It is also noteworthy that while each L7 load balancer works only for a specific application, our solution provides session persistence for all applications that make use of TLS.

III. Yuz: Session-Persistent Load Balancing

We propose an effective solution to session-persistent load balancing that is based on the session resumption mechanism of the TLS protocol. In the following, we first take a look at TLS and then present the method of Yuz.

A. Transport Layer Security

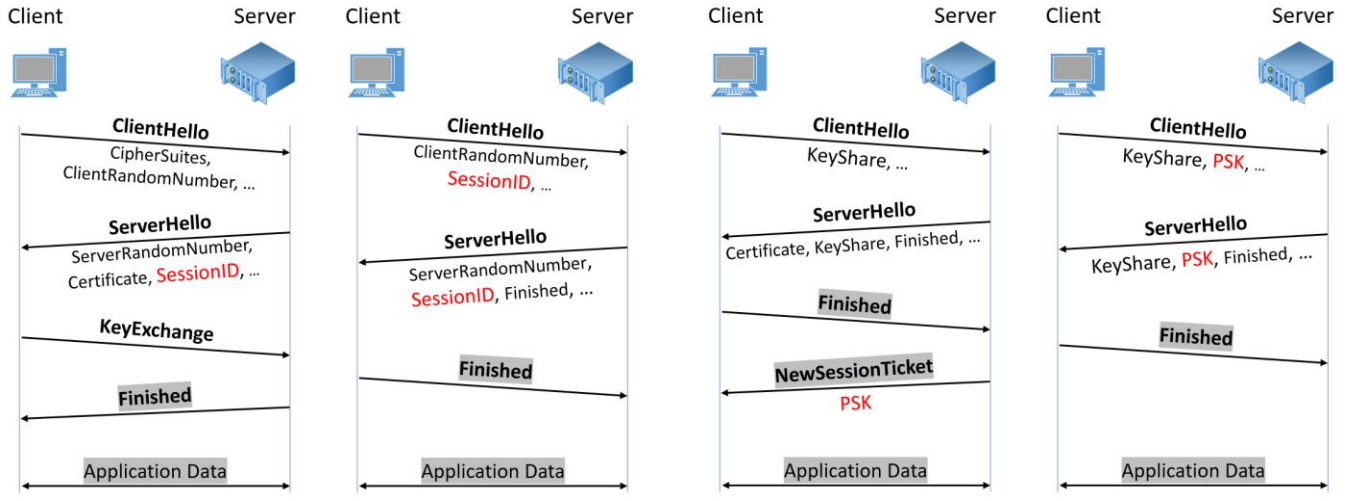
Transport Layer Security or TLS is the widely adopted security protocol of the Internet. It is a cryptographic protocol that adds a layer of security on top of reliable transport protocols (such as TCP) and provides end-to-end privacy and security of data sent between clients and servers over the Internet. TLS, which evolved from Secure Socket Layer (SSL), is used by many applications, such as Web (HTTPS), email, instant messaging, and VoIP. The most recent version of the protocol is TLS 1.3 [25], which was published in 2018. However, its previous version, TLS 1.2 [26], is still prevalent.

TLS defines and makes use of sessions to speed up the connection establishment procedure of consecutive connections between a client and a server. It should be noted that a TLS session differs from an application session, and a session-persistent load balancer aims at application sessions. However, as discussed in the following, the session resumption mechanism of TLS can be easily utilized to achieve an application-level session-persistent load balancer even without terminating TLS sessions at the load balancer.

Fig. 2 shows the TLS handshake protocol between a client and a server. The TLS handshake is the first step in establishing a TLS connection between a client and a server. During TLS handshaking, the client and the server exchange some messages to acknowledge and verify each other and agree on encryption algorithms and keys. This procedure adds considerable startup latency to the connections. To mitigate the latency cost of their subsequent connections, the client and server deploy a mechanism called *session resumption*. In TLS 1.2, after the initial full handshake (Fig. 2a), they store the negotiated session secret, i.e., the encryption algorithms and secret keys, along with the TLS session ID sent to the client by the server (within ServerHello message). The client includes the session ID in its first TLS message (ClientHello message) of subsequent TLS connection requests sent to the server. If the server recognizes the session, it replies with the same session ID, and they perform an abbreviated handshake as shown in Fig. 2b, and consequently, they reuse the previously negotiated session secret. Otherwise, the server may reply with a new session ID, and they carry out a full handshake. TLS 1.2 [26] has recommended an upper limit of one day for the session resumption lifetime.

An issue of the TLS session resumption by session IDs is that the server is required to deal with caching session secrets of many clients. Moreover, in the case of a cluster-based service, subsequent connections of the client are directed to different servers (especially because of the L4 load balancers), and servers that do not have the session secret cannot resume the TLS session, and they carry out a full handshake. To address this concern, TLS 1.2 also supports another kind of session resumption mechanism based on session tickets [27]. In the last step of handshaking, the server creates a ticket including the negotiated session secret encrypted with a secret key held only by the server itself and sends the ticket to the client. The client stores the ticket and includes it in the ClientHello message of subsequent TLS connection requests sent to the server. The server then decrypts the received ticket, recovers the session secret, and quickly resumes the session.

TLS 1.3 has replaced the two session resumption mechanisms with the concept of session resumption via Pre-



(a) Full handshake (TLS 1.2) (b) Session resumption (TLS 1.2) (c) Full handshake (TLS 1.3) (b) Session resumption (TLS 1.3)

Fig. 2. TLS handshake protocol. (Highlighted fields are encrypted messages)

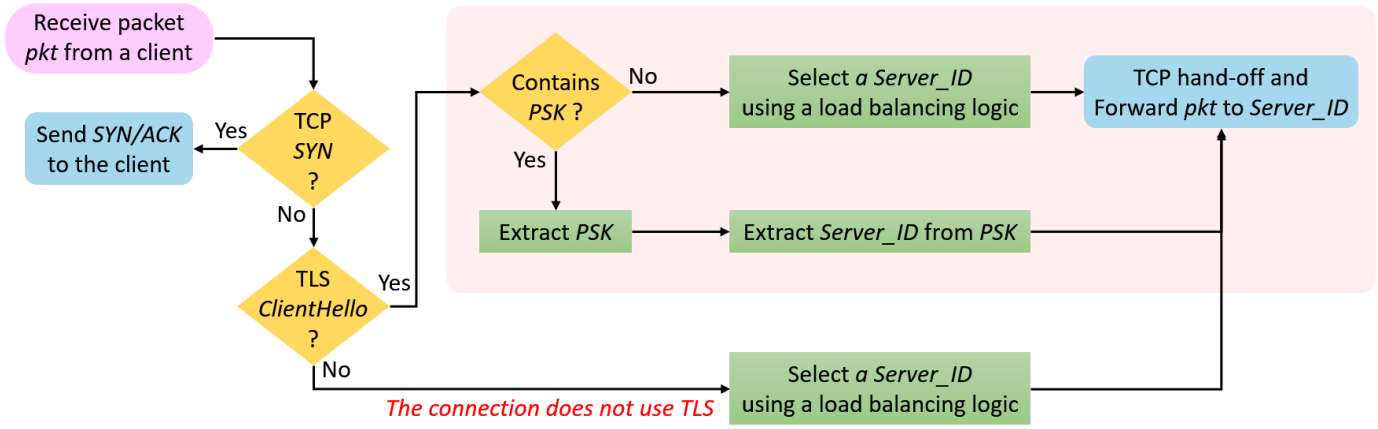


Fig. 3. The procedure of handling client packets by Yuz.

Shared Keys (PSK), which is similar to the ticket mechanism. In TLS 1.3, once a handshake has been finished, the server creates a PSK identity and sends it to the client (within NewSessionTicket message). The content of the PSK identity depends on the server and may include a database lookup key or a self-encrypted ticket. As with the two previous session resumption mechanisms, the client includes the PSK identity in the ClientHello message so the client and server can resume the session quickly.

B. Application-level session-persistent load balancing by utilizing TLS

The important points of the previous paragraphs can be summed up in two sentences: the server sends a piece of data as the TLS session identifier to the client, and the client includes the identifier in the first message of its subsequent connection requests. Hence, as long as a TLS session is valid, we can direct the subsequent connections of the client to the same server by utilizing the TLS session identifier coming with each connection request of the client. Since the client's requests are directed to the same server, we can achieve application-level session-persistent load balancing.

A naive approach to this problem is to make load balancers keep track of TLS sessions. To this end, a load balancer can look at the outgoing packets to find TLS messages carrying a session identifier (ServerHello in TLS 1.2 and NewSessionTicket in TLS 1.3) and consequently store TLS session identifiers issued by each server along with the server's ID in a session table. However, this approach suffers from the problems of stateful solutions, i.e., scalability, availability, and table insertion/lookup latency problems. Moreover, the NewSessionTicket message of TLS 1.3 is sent after finishing the handshake protocol. It means the message is encrypted, and therefore, a load balancer cannot read the included PSK of the message.

We conduct a stateless approach. A straightforward stateless solution is to make servers append their server identifier (for example, DIP) to the TLS session identifier they create and send to clients. Therefore, a load balancer can readily extract the server identifier from the session identifier of the ClientHello message and forward the subsequent packets of the connection to the server specified by the server identifier. However, this solution is prone to DDOS attacks. It allows users to realize if they are connected to the same server. They can wait to initiate their sessions to the same server and then suddenly increase the number and the load of their connections. To countermeasure this problem, Yuz employs a more robust technique. In the following, we discuss the method of Yuz in detail.

The method of Yuz. Fig. 3 represents the operations that Yuz carries out on packets received from a client. To make a new connection to the service, each client first sends a TCP connection request, i.e., a TCP SYN packet. The load balancer terminates each incoming TCP connection request by sending a SYN/ACK packet back to the client. Then, the client sends a TLS ClientHello message (typically as the payload of the third step of the TCP handshake, i.e., the ACK packet). The load balancer checks whether the ClientHello message contains a PSK[4] for resuming a TLS session. A ClientHello message that does not contain a PSK indicates that the client has not already established a TLS session or that the lifetime of its previous TLS session has expired. In this case, the load balancer selects a server according to a load balancing logic, such as Round Robin, least loaded, or random, and then transfers its endpoint of the connection along with the ClientHello message to the selected server using a TCP hand-off mechanism. We describe our TCP hand-off protocol and its advantages over Prism's mechanism later in this section. The server, which is now in charge of resuming the TCP connection with the client, creates a PSK and sends it to the client. Let S and P_n be the server's identifier and the PSK that a server normally creates for a TLS session, respectively. We make servers append their identifier to the normal PSK, which results in a new PSK, i.e., P_s:

$$P_s = append(P_n, S) \quad \text{(I)}$$

It can be easily done by modifying the TLS library of servers. To prevent clients from realizing the server's identity, the server makes P_s obscure using a simple encryption

[4] Considering the most recent version of TLS, the term "PSK" will be used to refer to the session identifier of a TLS session in the rest of this paper. However, it can be replaced by "session ID" or "session ticket" as for TLS 1.2 without any considerable difference in the implementations.

function and a secret key, K, which is held only by the servers and the load balancers.

$$P_e = encrypt(P_s, K) \quad \text{(II)}$$

As a result of the encryption, the part of P_e that represents the server identifier becomes dependent on P_n and K. The server sends P_e to the client as the PSK of the TLS session. Subsequent connection requests of the client will include the PSK. The load balancer extracts P_e that is included in the subsequent ClientHello messages, decrypts it using key K to obtain P_s, gets the server ID (i.e., S) from P_s, and directs the connections to the specified server. Therefore, Yuz maps all connections of each TLS session to the same server. It means the connections of each client are directed to the same server during its TLS session lifetime, and consequently, the server can keep application-layer session data in its local memory rather than the shared database. It should be noted that when a TLS session expires, the server writes the application session data into the shared database because a new TLS session of a client may be established with another server.

Connection hand-off. The connection hand-off mechanism of Yuz is simple and fast. Before presenting it, we review Prism's mechanism, the state-of-the-art TCP hand-off mechanism. Both methods are shown in Fig. 4. In the following discussion, we assume the simple network topology depicted in Fig. 5.

Prism is a reverse-proxy. Similar to most applications, Prism uses a TCP socket to establish a TCP connection with a client. After processing the incoming request at the application layer, Prism passes the socket to the application running on the chosen back-end server. Prism makes use of the TCP_REPAIR option and the connection serialization of Linux sockets to implement its TCP hand-off protocol. The load balancer puts its socket into repair mode, gets and serializes its state data such as buffer contents, sequence and ACK numbers, and TCP options, and sends the data to the application running on the chosen server. The application creates a new socket, puts it immediately into the repair mode, sets the data on the socket, and makes the socket resume its normal operation[5]. Moreover, since Prism is a reverse-proxy and processes incoming requests at the application layer, it also needs to hand off the TLS connection information and the application request to the back-end server. Before migrating the connection, the load balancer instructs the edge switches of the data center's network to block incoming packets of the connection (which is unlikely to happen). After the migration, the server instructs the switches to rewrite the destination IP and MAC addresses of the incoming packets of the connection to the server's addresses (DIP) and rewrite the source addresses of the outgoing packets of the connection to the load balancer's addresses (VIP). The address rewriting can be easily implemented by programmable switches.

Passing a socket to an application running on a back-end server must be done in the application layer, which results in

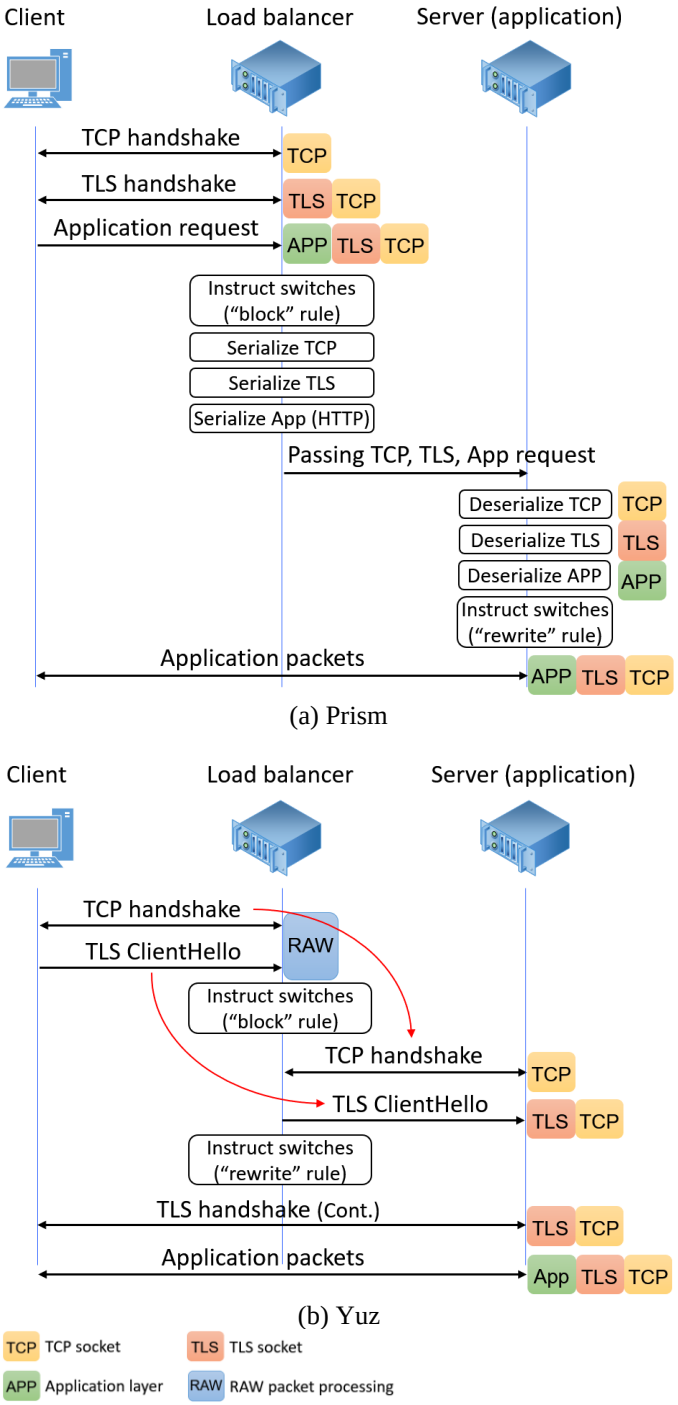


Fig. 4. Details of the hand-off mechanisms of Prism and Yuz.

an application-specific load balancer. It also means that each application (e.g, a Web application) that wants to deploy Prism needs to be modified in order that it includes the hand-off mechanism of Prism. Instead of the normal procedure of establishing a connection, the application must follow the mentioned steps to make a connection. Moreover, the state serialization/deserialization of TCP, TLS, and the application request in Prism takes a considerable processing time. As stated by the Prism paper, half of the time that is spent on the whole connection procedure is consumed by the TLS hand-off

[5] The serialization/deserialization procedure is done using TCP socket functions getsockopt(), setsockopt(), and connect().

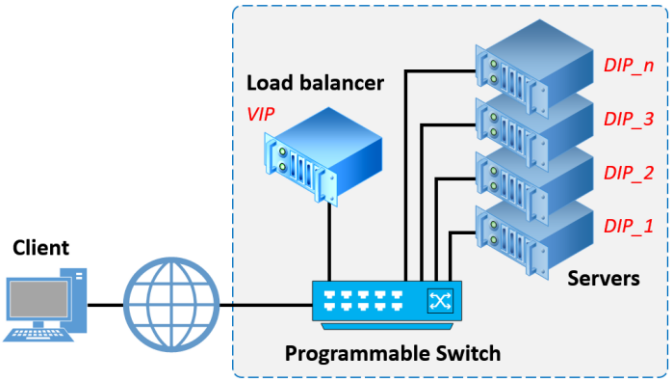


Fig. 5. Network topology

procedure [8]. On the other side, the load balancer machines of Yuz do not create and pass sockets, nor do they create and pass TLS sockets. Thanks to the early detection of the session, Yuz can replicate the TCP connection establishment procedure for the back-end server on behalf of the client. Hence, the server's application can normally establish TCP, TLS, and application-level connections.

Yuz load balancer does raw packet processing instead of using OS TCP sockets. Yuz responds to the TCP SYN packet of a client by sending a SYN/ACK packet. Then, the client sends the ClientHello packet. After processing the ClientHello packet and choosing a back-end server, the load balancer reproduces the TCP SYN packet of the client with destination MAC and IP addresses set to the server's addresses. As a result of receiving the SYN packet, the application running on the server, which is normally listening for a connection request, creates a socket to establish a connection with the client. In response to the packet, the server sends a SYN/ACK packet to the client. The switches are instructed to drop the SYN/ACK packet originating from the back-end server because it has already been sent by the load balancer to the client. Then, the load balancer sends the client's ClientHello packet (with destination addresses set to the server's) to the server. Finally, the load balancer instructs the switches to rewrite the IP and MAC addresses of the packets of the connection as stated before. We have used OpenFlow protocol for adding the address rewriting rules to the switches. From this point on, the client and the server can continue the connection.

The problem with this hand-off mechanism is the server-side TCP sequence number. At the time of TCP connection establishment, each side of the connection generates a random number as its initial sequence number. When the server creates a new socket, its initial sequence number is different from that of the load balancer. To resolve this problem, Yuz appends its initial sequence number to the payload of the TCP SYN packet that it sends to the server. We have made a slight modification to the implementation of the network stack (TCP library)[6] of the server in order that it uses the appended sequence number instead of a random number as the initial sequence number of new connections.

As can be seen, the complexity and cost of the hand-off mechanism of Yuz are much lower than the Prism's protocol. Since Yuz does not involve in establishing a TLS connection, processing an application request, and handing off them, it can hand off the established connection much faster than Prism. Moreover, Yuz stores neither session-to-server nor connection-to-server mappings. The fully stateless design of Yuz not only increases its performance but also provides a scalable solution. Since no session and connection state is stored in a load balancer, load balancer machines can be easily added or removed without any concern for ECMP remapping and consequently session persistence and connection affinity issues. Even though the connections of a session may initially go to different load balancers (by ECMP), they are ultimately directed to the same server by the load balancers.

C. Implementation

Implementing the proposed method entails two steps: making servers insert their identifier into PSK, and implementing a load balancer machine.

Servers. To make servers insert their identifier into PSK, we do not need to modify the services or applications such as Web, Email, FTP, or VoIP software. The applications typically make use of the TLS library of the server on which they run. Therefore, we need to modify the TLS library of servers to make them insert their identifier into PSKs that are sent to clients. OpenSSL [28] is a widely, if not the most widely, used TLS library which is open source. We have modified the OpenSSL library to add the mentioned feature to it. In the PSK generation function of the OpenSSL library, we added a small piece of code that appends the server's IP address (DIP) to the normally generated PSK, encrypts the combination of the PSK and DIP, and then sends the new PSK to the client. We have also added the reverse action into the function that processes the ClientHello message.

Load balancer machines. As stated before, Yuz does raw packet processing. We have implemented the load balancer using Data Plane Development Kit (DPDK) [29]. With the aid of DPDK, the load balancer software can benefit from low-cost IO processing for network packets and also bypass the OS kernel and achieve a high packet processing capacity. When a Yuz machine receives a ClientHello packet, it extracts the packet's PSK. Typically, ClientHello messages are small and each ClientHello can be accommodated in a single packet. Nevertheless, if a ClientHello message is transmitted via multiple packets (even due to fragmentation), the load balancer waits until the message is completed. After extracting the PSK, Yuz decrypts it using the secret key and gets the server identifier, which is the server's DIP in our implementation. If a ClientHello packet does not contain a PSK, the load balancer selects a server using a load balancing logic. Like Cheetah, the load balancing logic of Yuz does not depend on its other mechanisms, and any load balancing logic can be employed. However, in comparison with Cheetah, the load balancing decisions of Yuz is coarse-grained. While

[6] In Linux, the function tcp_v4_init_seq, which is located in tcp_ipv4.c file, generates the sequence number.

Cheetah can make a load balancing decision for each new connection, Yuz does it only for each new TLS session, which can result in load imbalances. The load imbalances are compared in the evaluation section. We have used a simple Round Robin strategy as the load balancing logic of our implementation.

IV. Evaluation

In the following, we present the evaluations we carried out. To compare Yuz with existing load balancing solutions, first, we measured the packet processing cost of each load balancer. Then, we evaluated the performance of a cluster-based service equipped with Yuz and compared it with other load balancers.

Testbed. Our testbed contains six machines for generating traffic workload, six machines as clustered servers running a service, a machine hosting the shared database of the service, and a machine running the load balancer. Each of the machines is equipped with an 8-core Intel Xeon 2690 CPU and 16 GB of RAM. All the machines are connected to an Arista 7050SX3-48C8 switch. While the machines that generate the traffic workload are connected to the switch using 10G links, all the other machines are connected to the switch via 100G links. It guarantees that the network links of the clustered servers will not become congested and they will not contribute to the tail latency. As stated before, this issue is orthogonal to our work [13]–[18].

We have installed eight virtual machines on each server of the cluster, and each virtual machine has its dedicated core. Therefore, we have a cluster of 48 virtual servers (and 48 DIPs) to run a cluster-based service. In addition to Yuz, we installed Cheetah [5], [21], Beamer [4], Nginx [9], and Prism [8] on the load balancer machine. Currently, the most high-performance L4 load balancer in the literature is Cheetah. Beamer is a stateless L4 load balancer with both software and P4 implementations. Nginx is the most popular commercial L7 load balancer, and Prism is the state-of-the-art L7 load balancer proposed in recent papers. All the load balancers were made to get the most out of the machine by assigning eight threads (workers) for each, which resulted in employing all eight cores of the machine. The load balancing logic of all the load balancers was set to Round Robin. It should be noted that all the existing L4 load balancers, including P4-based ones, do not provide session-persistent load balancing.

Service. The service that is run by the six clustered servers is an online shopping website. We have made the website using WordPress and WooCommerce. It makes use of application-layer sessions to store user-specific information such as cart items. It retrieves session data for each request. WordPress uses native PHP sessions. The native session management of PHP stores the session data locally on whatever server it is running on. Therefore, a session-persistent L7 load balancer such as Nginx is typically used with such clustered WordPress websites. To deploy an L4 load balancer, we need to store the session data in the database. The "WP session Manager" plugin is used to leverage the database in multi-server installations of WordPress. The plugin caches the session data of recently processed connections locally and also stores the session data in the database to deal with L4 load-balancing issues. In the case of the Prism load balancer, the web application does not work normally because it needs to support the hand-off mechanism of Prism. We modified the application in order that it gets incoming connections using the hand-off mechanism of Prism.

Traffic workload generation tool. There are various web server benchmarking tools that simulate many clients sending web requests. Nevertheless, a problem with most of them (as far as we examined) is that they cannot associate multiple requests of each simulated client with an application session. This is because they do not save the application-layer session ID of each client to include it as a cookie in the subsequent requests of the client. Some benchmarking tools, such as ApacheBench [30], have the option to stick to only one application session and associate all the generated requests to one session ID, which cannot fulfill our requirement. We need to simulate thousands of clients, each with its own application session, so all connections of each simulated client are associated with its application-layer session ID. To accomplish this, we employed a modified version of TRex [31]. TRex is an open-source traffic generator and benchmarking tool based on DPDK. It can generate stateless and stateful L3-7 traffic and emulate L7 applications such as HTTP/HTTPS. We modified the source code of TRex. The modified TRex saves the application session ID that each server issues to each simulated client and uses the session ID for all subsequent requests of the client. Moreover, we can set the number of session requests and also the interval between the requests of each session to a constant value or random. The traffic generator increases the request rate by increasing client sessions. It is noteworthy that the application-layer session ID of the Web is transferred via HTTP cookies.

A. Cost of packet processing

To measure the packet processing cost of each load balancer, we generate a traffic workload. Meanwhile, we measure the CPU cycles consumed by the load balancer software using *perf*, the profiler and performance analyzing

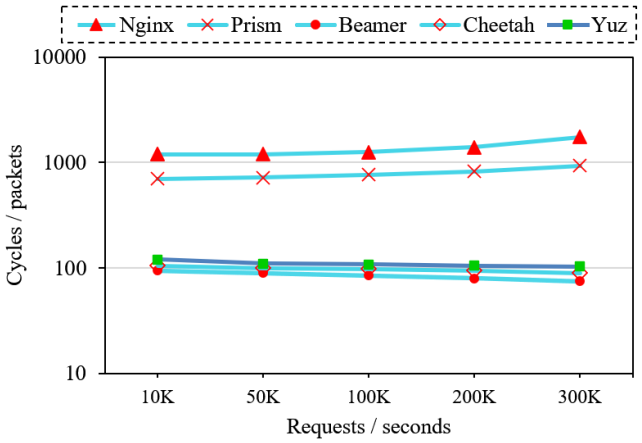


Fig. 6. Packet processing cost of different LBs in terms of CPU cycles per packet

tool of Linux. Fig. 6 shows the number of CPU cycles divided by the number of forwarded packets for different load balancers and different request rates. Note the logarithmic scale of the vertical axis. The packet processing cost of Yuz is about 10% higher than Cheetah and Beamer merely because of extracting PSK from the ClientHello message and the TCP hand-off procedure. These operations are done in the initial step of each new connection. It should be noted that extracting PSK is done for only one packet of each connection. As can be seen in Fig. 6, Nginx and Prism, which are L7 load balancers, consume an order of magnitude more cycles than Yuz and Cheetah because they act as a reverse-proxy. The processing cost of Prism is lower than Nginx because it does not relay traffic after handing off the connection. It is noteworthy that we had also disabled all unnecessary features of Nginx, and it was doing a simple load balancing task. Interestingly, the processing cost of Nginx and Prism increases as the request rate increases. As a result of their stateful design, an increase in the request rate results in more populated and larger state tables and consequently higher insertion and lookup time.

We have also compared the hand-off mechanisms of Yuz and Prism by measuring the time the load balancer and servers spend migrating a connection. While Prism's connection hand-off takes 350 µs, Yuz spends 135 us to migrate a connection. The higher connection hand-off cost of Prism is due to the serialization and deserialization of TCP, TLS, and the application request. As reported by the Prism paper [8], half of the time it spends handing off a connection is consumed by the TLS serialization and deserialization tasks, which are not carried out in the Yuz scheme.

B. Performance of the clustered system

In the second experiment, we evaluated the performance of the clustered system with each load balancer. Unlike the evaluations of all of the previous load balancing studies, which solely measure the throughput and latency of load balancers by generating independent requests, our evaluation investigates a more realistic scenario by taking into account application sessions and measuring the performance of the clustered system.

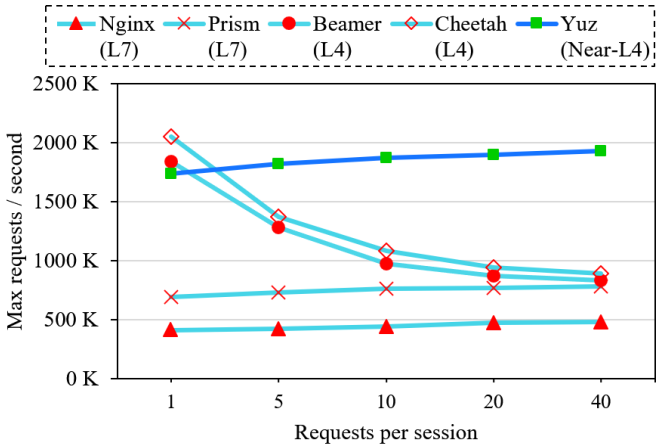


Fig. 7. Maximum request rate handled by the clustered servers for different LBs.

Achieved request rate. First, we measured the maximum request rate that can be handled by the clustered system provided with each of the load balancers. We performed this evaluation with different number of requests per session. The interval between each session's requests is 1 second.

Fig. 7 shows the achieved request rate for different load balancers as a function of number of requests per session. The first interesting point is that the L4 load balancers demonstrate their best performance when each request implies a new session, which is not the case for most applications. This point has not been taken into account in any of the previous studies. As the ratio of requests to sessions increases, the efficiency of existing L4 load balancers decreases. A higher ratio of requests per session means that a higher proportion of requests are dependent on session data. The back-and-forth transfer of application-layer session data for these requests limits the request processing capability of the clustered server. However, when the clustered servers make use of session-persistent load balancers, their performance is not affected by session-dependent requests.

Nginx and Prism, the L7 load balancers, resulted in a significantly lower request rate than others. The heavy processing cost of L7 load balancers greatly limits the request rate that can be delivered to the clustered servers. Yuz, however, provides a session-persistent yet high-throughput load balancing solution. As a result of its session-persistent load balancing, each server of the cluster can cache application-layer session data and process requests of each session immediately.

CPU utilization. To shed more light on the issue, we measured the CPU utilization of the load balancer machine, the clustered servers, and the shared database machine at the maximum request rate that can be achieved by each load balancer. Fig. 8 shows the results. It confirms that in the case of L7 load balancing, the load balancer becomes the system's bottleneck and does not allow the cluster to receive requests as high as its maximum capacity. Cheetah and Beamer, the L4 load balancers, have higher throughput than L7 load balancers and deliver more requests to the cluster, which leads to higher cluster utilization. However, they significantly increase the

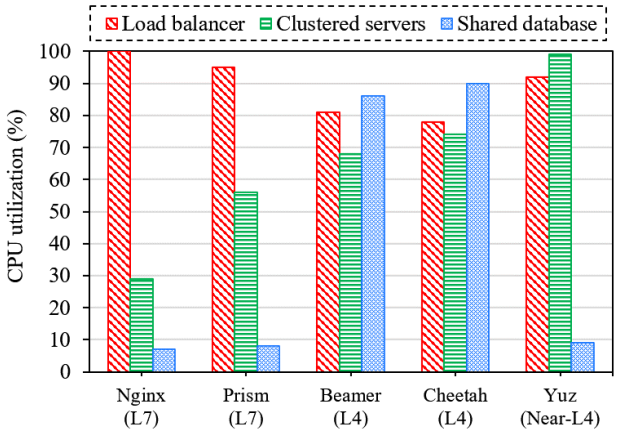


Fig. 8. CPU utilization of machines at the maximum request rate that can be achieved by each LB.

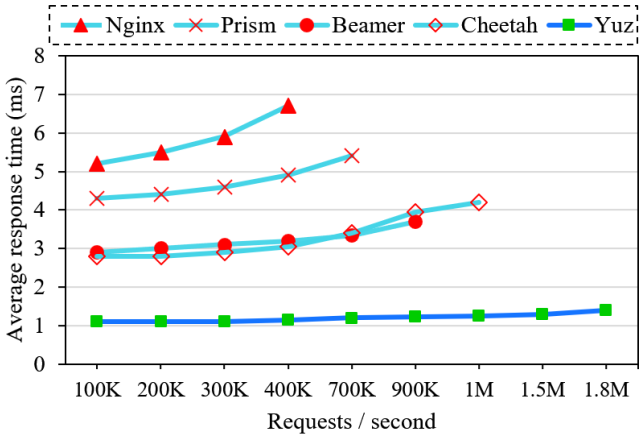


Fig. 9. Average response time of the clustered servers for an increasing number of requests per second.

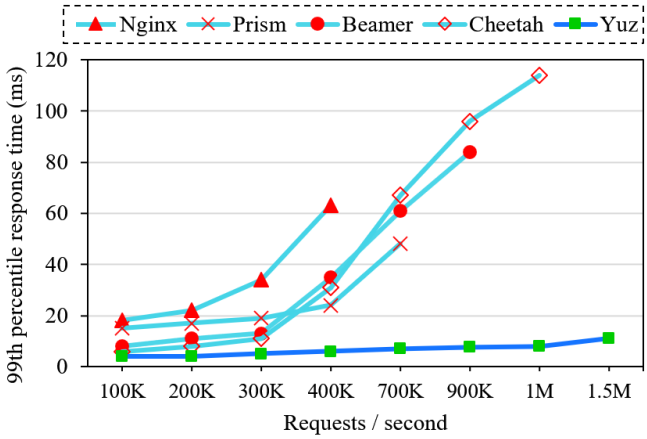


Fig. 10. 99th percentile response time of the clustered servers for an increasing number of requests per second.

load of the shared database due to the frequent requests for the session data. Moreover, part of the cluster utilization is due to session data retrieval. Yuz causes the cluster to achieve its maximum utilization, which is a more effective utilization because a significantly lower CPU time is wasted on session data retrieval.

Response time. As mentioned before the response time of servers directly affects user experiences. Fig. 9 shows the average response time of the clustered system for an increasing number of requests per second. Yuz outperforms other load balancers. Not only is the response time of the system equipped with Yuz significantly lower than others but also it is not considerably affected by increasing the request rate. On the other side, when the system uses L7 load balancing, its response time increases as the request rate increases because an increase in the request rate results in larger state tables and higher insertion and lookup time in the load balancer. As for Cheetah and Beamer, the rise in the response time stems from the increase in the rate of session data transfers between the servers and the shared database which means that the requests have to queue for a longer time. We have also measured the 99th percentile of the response time, which is illustrated in Fig. 10. The interesting point is that as the request rate for Cheetah approaches its maximum capacity, its response time tail increases significantly and it becomes worse than other load balancing schemes. The result is almost similarly observed for Beamer as well. It means that while the average response time of existing L4 load balancers is lower than L7 ones, their long response time tail can result in bad user experiences. Therefore, to prevent long tail latency, data centers need not only network load balancers [13]–[18] but also effective server load balancing solutions.

Size of session data. The size of session data is dependent on user requests and also the application. In the mentioned experiments, the size of session data was about 1 MB. To evaluate different application workloads, we repeated the experiment using different sizes of session data. We defined session data of different sizes in the application and arranged the request so that they do not considerably increase the session data. Fig. 11 shows the maximum request rate achieved by each load balancer for different sizes of session data. While increasing the size of session data does not considerably affect the performance of the cluster servers equipped with session-persistent LBs, it significantly degrades the performance in the case of L4 load balancers because it takes a longer time for larger session data to be transferred between the clustered servers and the shared database. Moreover, larger session data results in a higher cache miss rate in the shared database and consequently higher retrieval time. Interestingly, Cheetah and Beamer fall behind the L7 load balancers for session data larger than 10 MB.

Load imbalance. As mentioned in the previous section, although Yuz can employ any load balancing logic, its load balancing decisions are coarse-grained in comparison to Cheetah because it makes load balancing decisions only for new TLS sessions. We cannot evaluate the load imbalance using a workload similar to the previous experiments. If we use constant values for the parameters, i.e., the number of each session's requests and the interval between each session's requests, the Round Robin strategy of Cheetah and Yuz results in a uniform distribution of connections among the cluster's

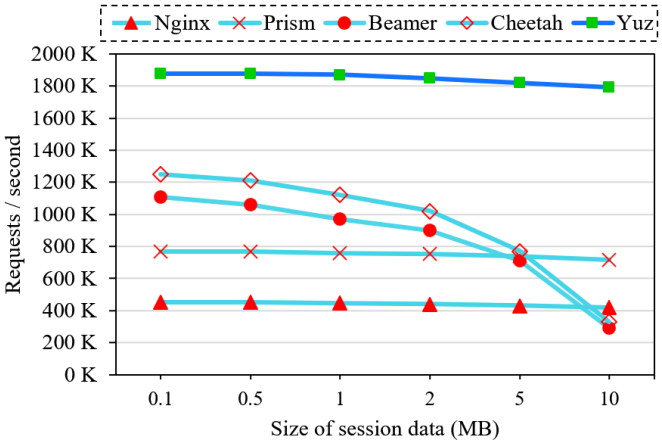


Fig. 11. Maximum request rate achieved by each load balancer for different sizes of session data.

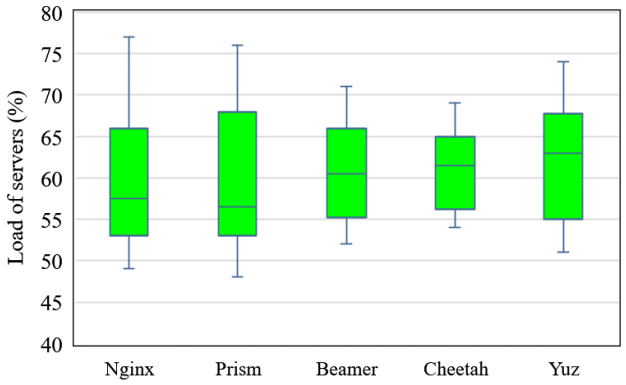


Fig. 12. Distribution of loads of servers for different load balancers.

servers. To make the conditions more realistic, we set the value of the mentioned parameters to random. We repeated the experiment with different values of the parameters to achieve an average cluster load of 60% and measured the load of each virtual server in that condition. Similar to Cheetah and Maglev, we consider the number of active connections of a server as its load. For the case of each load balancer, the load of each server is divided by the number of maximum connections achieved by that load balancer. Fig. 12 shows the distribution of the measured loads. As can be seen in the figure, although the load imbalance of Yuz is higher than Cheetah, it does not result in a very unbalanced load.

V. Discussion

In the following, we discuss some important issues regarding the proposed method.

A. Applicability

The first issue regarding the proposed method is that it only works for services that employ TLS. However, this requirement does not put an obstacle in the way of the proposed method because nowadays most Internet-based services use TLS, and especially, TLS is critical for the applications that need to recognize clients using sessions. Nevertheless, there may be traffic and application that do not use TLS. In this case, Yuz functions as an L4 load balancer similar to previous ones, which do not provide session-persistence. For each new connection request that does not include a TLS ClientHello message, Yuz instructs the switches to forward the connection to a specific server selected by a load balancing logic (using the hand-off mechanism); and the session data, if any, must be kept and retrieved using the shared database. The other requirement is that both the client and server must support the TLS resumption mechanism. By default, the mechanism is active in all popular browsers, and they use it. The TLS resumption lifetime is also important. On the server side, this parameter can be set to, for example, one day. TLS 1.2 [26] has recommended an upper limit of one day for the session resumption lifetime, while TLS 1.3 [25] has recommended an upper limit of seven days.

It is noteworthy to point out some studies on TLS adoption. In 2018, a study [32] examined the Alexa Top Million. According to the reported results, about 70% of the sites supported TLS, and among them, 95.9% supported TLS session resumption. The TLS lifetime of more than 50% of the sites was higher than one day. The other sites used a lifetime in order of hours. The study also showed that the browsers Chrome, Firefox, Edge, Opera, and Safari had used a default configuration of 60 min, one day, 600 min, 30 min, and 120 min for the TLS lifetime, respectively. Another recent study [33] shows that more than 90% of Alexa Top 1K have adopted TLS. In regard to TLS 1.3, a study has revealed that after the TLS 1.3 standardization, it has been adopted at a higher rate than TLS 1.2 [34].

B. Implication of modifying the server-side libraries

Another issue that may be a cause of concern is the effect of the modifications made to the TCP and TLS libraries of the servers. We emphasize that we have not modified the protocols. The modifications are slight changes in the server-side implementation of the protocols without deviating from the protocol standards, and the modifications do not have any impact on the functionality and applicability of the protocols. From the clients' perspective, nothing has changed. Regarding TCP, we have modified the function that generates the initial sequence number of each new connection (tcp_v4_init_seq). Normally, this function generates a random number. We have added a piece of code to this function wherein, if the SYN packet sent by the load balancer contains a sequence number in its payload (which is the random sequence number generated by the LB), it will utilize that sequence number instead of generating a new one. Hence, it does not affect the functionality of TCP. The modification we made in TLS is in PSK generation function. Our modification, which is placed in the end of the function, involves appending the server identifier (DIP) to the end of the original PSK and encrypting the result using a key that is exclusive to servers and load balancers. We have also added the reverse of this procedure in the beginning of the session resumption function. Therefore, this modification neither affects the functionality of TLS nor causes any security implications. A potential security implication could have been if clients were able to determine that they were connected to the same server (and launch a DDoS attack). However, by encrypting the combination of the original PSK and the DIP, we arrive at a random number that is dependent on both the DIP and the original PSK. Therefore, it is impossible for clients to identify the servers. Nevertheless, malicious clients may use the same P_e to reach to the same server and put it under pressure. This problem can be easily resolved. PSK is a unique identifier that a server issues for each client. There is no reason for different clients to have the same PSK. Therefore, we can address this issue by having the load balancer block requests from different clients that have the same PSK. This task can also be offloaded to the network's firewall.

C. The merit of the proposed solution

It would be ideal to have an original and optimized protocol specifically designed for this purpose rather than modifying existing protocols. However, designing a new protocol will not easily solve this problem in practice, as the protocol needs to be installed on both the client and server sides, and we do not have access to the clients to make them use the new protocol. In other words, designing a new protocol requires standardization and widespread adoption. Therefore, some solutions that are proposed in the literature [4], [8], [21] choose to utilize existing protocols and make slight modifications on the server side to become practical. The positive point of our solution is that unlike some other methods like Prism, it does not require modifying applications (e.g., web). It only needs some minor modifications to the server-side implementation of the lower common layers (TCP and TLS), which can be easily deployed by a simple patch on Linux servers. With Yuz, any application installed on the servers can benefit from the session persistence capability, which makes Yuz suitable for cloud environments.

Finally, we stress that the proposed load balancer is effective for clustered servers that deal with application sessions and suffer from the back-and-forth transfer of application-layer session data for each request. For services that need frequent communication of back-end servers, such as MapReduce, and also for multi-tier services, the benefits from a session-persistent load-balancing may not be significant.

VI. Conclusion

In this paper, we presented Yuz, a session-persistent load balancer for clustered servers of data centers. The proposed method works near layer 4 and does not act like a reverse-proxy. Hence, it has a traffic processing capacity as high as layer-4 load balancers. Thanks to the fully stateless design of Yuz, it can be easily deployed in a scalable manner while preserving both connection affinity and session persistence. For the first time in the literature, our evaluation took into account application sessions. The evaluation showed that the performance of cluster-based services that make use of Yuz is significantly higher than the ones that employ other load balancing approaches. Furthermore, Yuz significantly reduced the average and tail of response time. Our future work will focus on addressing UDP and QUIC protocols by making use of Datagram Transport Layer Security (DTLS).

References

[1] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, and others, "Ananta: Cloud scale load balancing," *ACM SIGCOMM Computer Communication Review*, ACM New York, NY, USA, vol. 43, no. 4, pp. 207–218, 2013.

[2] 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 *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 523–535, March 2016.

[3] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford, "Efficient traffic splitting on commodity switches," in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pp. 1–13, December 2015.

[4] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless datacenter load-balancing with beamer," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 125–139, April 2018.

[5] T. Barbette, C. Tang, H. Yao, D. Kostić, G. Q. Maguire Jr, P. Papadimitratos, and M. Chiesa, "A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 667–683, February 2020.

[6] M. Meiss, J. Duncan, B. Gonçalves, J. J. Ramasco, and F. Menczer, "What's in a Session: Tracking Individual Behavior on the Web," in *Proceedings of the 20th ACM conference on Hypertext and hypermedia*, pp. 173–182, June 2009.

[7] R. Gandhi, Y. C. Hu, and M. Zhang, "Yoda: A highly available layer-7 load balancer," in *Proceedings of the Eleventh European Conference on Computer Systems*, pp. 1–16, April 2016.

[8] Y. Hayakawa, M. Honda, D. Santry, and L. Eggert, "Prism: Proxies without the Pain," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 535–549, 2021.

[9] "Nginx - Web server and load balancer." [Online]. Available: www.nginx.org.

[10] "HAProxy - HTTP load balancer." [Online]. Available: www.haproxy.org.

[11] M. Kogias, R. Iyer, and E. Bugnion, "Bypassing the load balancer without regrets," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, pp. 193–207, October 2020.

[12] S. M. Hosseini, A. H. Jahangir, and S. Daraby, "Session-persistent Load Balancing for Clustered Web Servers without Acting as a Reverse-proxy," in *17th International Conference on Network and Service Management (CNSM)*, pp. 360–364, October 2021.

[13] S. Ghorbani, Z. Yang, P. B. Godfrey, Y. Ganjali, and A. Firoozshahian, "Drill: Micro load balancing for low-latency data center networks," in *31st ACM SIGCOMM Conference* , pp. 225–238, August 2017.

[14] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, "CONGA: Distributed Congestion-Aware Load Balancing for Datacenters," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pp. 503–514, August 2014.

[15] M. Shafiee and J. Ghaderi, "A Simple Congestion-Aware Algorithm for Load Balancing in Datacenter Networks," *IEEE/ACM Transactions on Networking*, vol. 25, no. 6, pp. 3670–3682, December 2017.

[16] Q. Shi, F. Wang, and D. Feng, "IntFlow: Integrating Per-Packet and Per-Flowlet Switching Strategy for Load Balancing in Datacenter Networks," *IEEE Transactions on Network and Service Management*, vol. 17, no. 3, pp. 1377–1388, September 2020.

[17] F. Wang, H. Yao, Q. Zhang, J. Wang, R. Gao, D. Guo, and M. Guizani, "Dynamic Distributed Multi-Path Aided Load Balancing for Optical Data Center Networks," *IEEE Transactions on Network and Service Management*, vol. 19, no. 2, pp. 991–1005, June 2022.

[18] J. Cao, R. Xia, P. Yang, C. Guo, G. Lu, L. Yuan, Y. Zheng, H. Wu, Y. Xiong, and D. Maltz, "Per-packet load-balanced, low-latency routing for clos-based data center networks," in *Proceedings of the ninth ACM conference on Emerging*

networking experiments and technologies (CoNEXT), pp. 49–60, December 2013.

[19] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang, "Duet: Cloud scale load balancing with hardware and software," *ACM SIGCOMM Computer Communication Review*, ACM New York, NY, USA, vol. 44, no. 4, pp. 27–38, 2014.

[20] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pp. 654–663, May 1997.

[21] T. Barbette, E. Wu, D. Kostic, G. Q. Maguire, P. Papadimitratos, and M. Chiesa, "Cheetah: A High-Speed Programmable Load-Balancer Framework with Guaranteed Per-Connection-Consistency," *IEEE/ACM Transactions on Networking*, Institute of Electrical and Electronics Engineers Inc., vol. 30, no. 1, pp. 354–367, 2022.

[22] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pp. 15–28, August 2017.

[23] C. Zeng, L. Luo, T. Zhang, Z. Wang, L. Li, W. Han, N. Chen, L. Wan, L. Liu, and Z. Ding, "Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 1345–1358, April 2022.

[24] M. E. Haque, Y. hun Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley, "Few-to-many: Incremental Parallelism for Reducing Tail Latency in Interactive Services," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 161–175, May 2015.

[25] E. Rescorla, "The transport layer security (TLS) protocol version 1.3, RFC8446," 2018. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8446.

[26] T. Dierks and E. Rescorla, "The transport layer security (TLS) protocol version 1.2, RFC5246," 2008. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5246.

[27] J. Salowey, "Transport Layer Security (TLS) Session Resumption without Server-Side State, RFC5077," 2008. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5077.

[28] "OpenSSL - Open source SSL/TLS library." [Online]. Available: https://www.openssl.org.

[29] "DPDK: Data Plane Development Kit." [Online]. Available: https://www.dpdk.org.

[30] "ApacheBench." [Online]. Available: http://httpd.apache.org/.

[31] "TRex Traffic Generator." [Online]. Available: https://trex-tgn.cisco.com/.

[32] E. Sy, H. Federrath, C. Burkert, and M. Fischer, "Tracking users across the web via TLS session resumption," in *Proceedings of the 34th Annual Computer Security Applications Conference*, pp. 289–299, December 2018.

[33] H. Lee, D. Kim, and Y. Kwon, "TLS 1.3 in practice: how TLS 1.3 contributes to the internet," in *Proceedings of the Web Conference 2021*, pp. 70–79, April 2021.

[34] R. Holz, J. Hiller, J. Amann, A. Razaghpanah, T. Jost, N. Vallina-Rodriguez, and O. Hohlfeld, "Tracking the deployment of TLS 1.3 on the web: a story of experimentation and centralization," *ACM SIGCOMM Computer Communication Review*, vol. 50, no. 3, pp. 3–15, July 2020.

Mohammad Hosseini received his M.Sc. and Ph.D. degrees in Computer engineering from Sharif University of Technology, Tehran, Iran, in 2014 and 2019, respectively. He is currently an assistant professor at the Faculty of Computer Science and Engineering, Shahid Beheshti University, where he carries out research into high-performance data center networking.

Sina Darabi received his BSc and MSc degrees in Computer engineering from Isfahan University of Technology, Isfahan, Iran, in 2012 and 2015, respectively. He received his PhD degree in Computer engineering from Sharif University of Technology, Tehran, Iran, in 2022. He has published numerous papers in top-tier conferences such as MICRO, ISCA, and SIGMETRICS. He is currently a postdoctoral researcher at the Institute for Research in Fundamental Sciences (IPM), Tehran, Iran. His field of interest is high performance computer systems.

Amir Hossein Jahangir obtained his PhD. in Industrial informatics from Institut National des Sciences Appliquées, Toulouse, France, in 1989. Since then, he has been with the Department of Computer Engineering, Sharif University of Technology, Tehran, Iran, and has served during his career as the Head of the Department, Head of Computing Center, and is now an associate professor and the director of Network Evaluation and Test Laboratory, an accredited recognized laboratory in the field of network equipment test. His fields of interest comprise network equipment test and evaluation methodology, network security, and high-performance computer architecture.

Ali Movaghar (Senior Member, IEEE) received the M.S. and Ph.D. degrees in computer, information, and control engineering from the University of Michigan, Ann Arbor, MI, USA, in 1979 and 1985, respectively. He is currently a Professor with the Department of Computer Engineering, Sharif University of Technology, Tehran, Iran. His research interests include performance or dependability modeling and formal verification.