Session-persistent Load Balancing for Clustered Web Servers without Acting as a Reverse-proxy

S. Mohammad Hosseini
Department of Computer Engineering
Sharif University of Technology
Tehran, Iran
smhoseini@ce.sharif.ir

Amir Hossein Jahangir
Department of Computer Engineering
Sharif University of Technology
Tehran, Iran
jahangir@sharif.ir

Sina Daraby
Department of Computer Engineering
Sharif University of Technology
Tehran, Iran
daraby@ce.sharif.ir

***Abstract*— A Load balancer is an essential component of a clustered web-server system which distributes requests among servers. Layer-4 load balancers process requests faster than Layer-7 ones, but they cannot provide session-persistent load balancing. On the other side, layer-7 load balancers, which are known as reverse-proxies, can direct all requests of an application-level session to the same server, but they impose a high processing cost due to processing requests at the application level. We propose a session-persistent load balancing approach which uses TLS session data instead of processing at the application level. It outperforms existing approaches in terms of transaction rate and response time.**

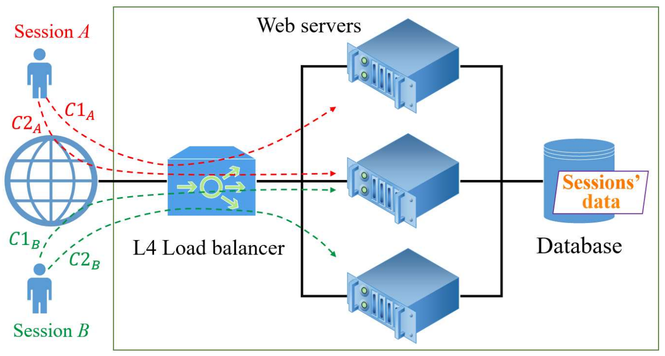
***Keywords*—Load balancer, Web server, Session persistence, HTTP, TLS.**

I. Introduction

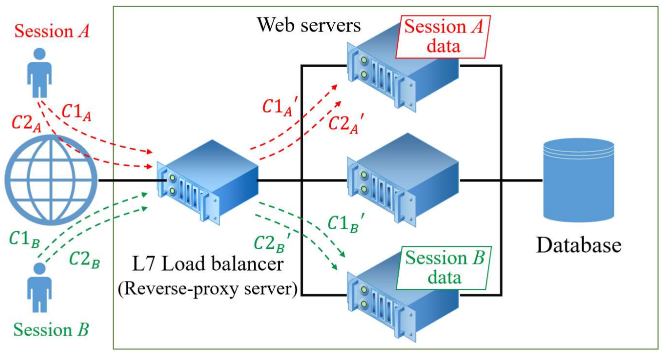
Data enters rely on load balancers to distribute clients' requests among servers and cope with a huge amount of traffic load. Considering the OSI protocol stack layer at which a load balancer operates, there are two load balancing approaches: Layer-4 (L4), and Layer-7 (L7) load balancing.

Layer-4 load balancers 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 persistence* (also known as *connection affinity*), i.e. directing packets belonging to the same connection to the same server. When an L4 load balancer receives a TCP SYN packet, it selects one of the servers based on a load balancing logic, such as Round Robin or random, 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. An important problem associated with 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 visit to a website. This can include filling out a series of forms in a website, adding items to a shopping cart, or checking a mailbox and sending emails from an account. The transactions result in many network-level connections established and closed during the time the session is open. A session can last from a few minutes to several hours and even days. The important point is that sessions cannot be recognized using network-level information such as IP addresses and port numbers because clients may be behind a NAT/proxy, a client's IP may change due to mobility, or TCP connections of a session may use different source ports.

As a result of session-unaware load balancing, the TCP connections of an application session are directed to different servers while the session is open. 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, such as cart items in an online shopping application, must be saved in a shared database as shown in Fig. 1a. When a server receives a connection request, the server processes the request up to the application layer and determines its session. Then, the server gets the session data from the shared database (if the request is related to an open session) and saves all modifications to them in the database to be used by the next connections of the session which may be served by other servers. On the other side, L7 load balancers consider and analyze application-level information of requests. One of the most beneficial abilities that can be achieved by L7



(a) L4 load balancing



(b) L7 load balancing

Fig. 1. Clustered web-server system 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.

load balancing is *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. As a result of having a session-persistent load balancer, 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) and consequently process subsequent requests of the session faster. To achieve this ability, existing L7 load balancers act as a *reverse-proxy*. A reverse-proxy terminates each incoming request (for example, connection $C1_A$ in Fig. 1b) up to the application-level and processes information such as HTTP cookies to identify its session. After selecting its appropriate server, the load balancer makes a new connection ($C1_A'$) to the selected server and transfers data between the client and the server.

However, the session persistence ability of existing load balancers does not come for free. Taking a web application as an example, a load balancer becomes a complete web server when it acts as a reverse-proxy. The reverse-proxy must (1) terminate each incoming TCP connection, (2) perform Transport Layer Security (TLS) handshake and decrypt the connection payload, (3) receive and process the HTTP request and extract its session ID from cookies, (4) make a new connection to the corresponding server, (5) send the request to the server, (6) receive the server response, (7) encrypt the response, (8) and send the response back to the client. This procedure imposes a heavy processing cost and significantly limits the request (transaction) rate at which the clustered web server system can service. Hence, the clustered web server systems that deal with millions of requests per second adopt L4 load balancing (such as the Maglev solution [1] which is used by Google) and handle clients' intermediate information using a shared database.

In this paper, we propose a session-persistent load balancer without acting as a reverse-proxy so that we can achieve a high-performance clustered web server system. The proposed solution performs load balancing without processing information in the application layer or even terminating TLS sessions. We use TLS session data to identify client sessions. In other words, we add the session persistence feature to L4 load balancers. Our evaluation shows that the proposed method outperforms existing load balancing approaches in terms of transaction rate and response time.

The rest of this paper is organized as follows: in Section II we briefly review previous approaches to L4 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. Literature review

Load balancing has been an active area of research since the late 1990s, when the Internet faced an explosive growth of traffic on the World Wide Web [2]. Software and hardware scale-up solutions could not keep pace with the ever-increasing demand placed on popular web-based services. As a locally scale-out solution, cluster-based web systems emerged [3]. A cluster-based web system consists of a collection of servers tied together in a data center to act as a single entity serving a web service. A front-end node called load balancer receives the incoming requests destined to the web system and distributes them among servers.

Traditional load balancers were implemented as dedicated hardware devices, and they were not scalable. The need for scalable and high available load balancing in large data centers led researchers to develop distributed load balancers. A challenging problem of distributed L4 load balancers is to provide connection persistence, especially when the set of load balancers or back-end servers changes due to failure, upgrading, or adding/removing 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.

One of the first load balancing solutions that tried to address this problem is Ananta [4], a stateful distributed software load balancer which runs on commodity hardware. In this solution, numerous load balancers can be easily deployed for a service in a data center. Ananta relies on border routers and Equal-cost multi-path routing (ECMP) to split incoming traffic evenly among load balancers. When a TCP SYN packet arrives at an Ananta load balancer, it randomly chooses one of the servers and forwards the packet to the server. Moreover, the load balancer stores the 5-tuple of the packet and the chosen server in a local connection table to be used for directing the subsequent packets of that connection to the same server. However, the solution falls short in terms of connection persistence when the pool of load balancers changes. ECMP uses a simple hash mechanism to distribute traffic among next hops (hash of IP addresses modulo the number of next hops). Therefore, if the set of load balancers changes, packets of ongoing connections are forwarded to load balancers that do not have any state for them, which in turn results in forwarding packets to wrong servers and breaking connections [5].

Niagara [6] utilizes SDN switches and an Open-flow controller to implement a scalable load balancing system. The centralized controller can share the connection states among load balancers, and consequently, if a packet is forwarded to a wrong load balancer by ECMP, the controller can provide the load balancer with the correct server of the packet. However, the controller not only adds latency overhead but also negatively affects forwarding performance.

Maglev [1] is another stateful distributed software load balancer which tries to minimize connection disruptions when the pool of load balancers or the set of back-end servers change. To this end, Maglev makes use of a hash method named “Maglev hashing” to map packets to servers. Maglev hashing, which is an improved version of consistent hashing [7], provides two important properties. First, it evenly splits new connections among back-end servers. Second, if the set of back-end servers changes, packets of connections will likely be mapped to the same server as they were before. In other words, both the load balancing logic and the connection persistence mechanism are carried out by the hashing method. Since the hashing method does not guarantee connection persistence, Maglev also maintains a connection table. When a Maglev load balancer receives a packet for which it does not have a state in its local connection table, it determines the destination server using Maglev hashing and stores the decision in the connection table.

Beamer [8] is a stateless load balancer which does not use a connection table. Similar to Maglev, its load balancing logic relies on an improved version of consistent hashing, but it does not store per-connection state. Beamer uses a hashing method named “Stable hashing” which works better in terms of uniform distribution. However, Stable hashing does not guarantee connection persistence as well, and when the set of load balancers or back-end servers changes, some packets may be forwarded to wrong servers. Beamer deals with the problem using a mechanism named “daisy chaining”. In this mechanism, if a server receives a packet for which it does not have a state, the server forwards the packet to another server.

Cheetah [9] is a new distributed load balancer which does not use the connection table as well. To provide connection persistence, Cheetah encodes the identifier of the selected server into the TCP timestamp options field of packets directed towards the client. On the other side, the client is expected to include the identifier in the subsequent packets that it sends towards the server. 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.

All the mentioned researches have focused on distributed L4 load balancers. Their goal is to provide a better L4 load balancing system in terms of scalability, availability, uniform distribution, and connection persistence. However, none of current L4 load balancers can provide session persistence. Currently, session persistence load balancing can be provided by L7 load balancers. Nginx [10] and HAProxy [11] are the most well-known L7 load balancers for web applications. They are web server software, and they process requests at the application layer. Hence, they are significantly worse than L4 load balancers in terms of latency and throughput. Moreover, their availability is limited and they exhibit a single point of failure. If a layer-7 LB fails, all connections handled by that LB are reset. Yoda [12] is a high-available L7 LB which tries to resolve this problem by storing flow states in a persistent storage. However, as with other L7 load balancers, its performance is lower than L4 LBs.

III. Proposed Method

We propose a simple but effective solution to L4 session-persistent load balancing which is based on TLS session resumption. 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 we discuss in the following, TLS session data can be easily utilized to achieve an application-level session-persistent load balancer even without terminating TLS sessions at the load balancer.

Before discussing the proposed method, we glance at TLS 1.2 handshake protocol [13] between a client and a server which is shown in Fig. 2. While establishing a new TLS session (Fig. 2a), the client and the server exchange some messages to acknowledge and verify each other and agree on encryption algorithms and session keys (which are called session secret). We do not go into detail about the protocol, but the important point for us is that the server assigns a session ID to the TLS session and sends the ID to the client before the encryption starts. The client and the server save the negotiated TLS session secret along with the session ID for a specified time duration. The client includes the session ID in its subsequent TLS connection requests to the server. If the server recognizes the session, it replies with the same session ID so they can perform an abbreviated handshake as shown in Fig. 2b. If the server is not willing to resume the TLS session (for example due to time expiration), the server replies with a new session ID, and consequently, a full handshake is performed.

The session ID of each TLS session in the set of active TLS sessions of a server must be unique. Servers assign a long and random string of bytes as session ID (32 bytes in TLS 1.2) to each client. 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 client’s TLS session ID coming with each connection request. As the clients’ requests are directed to the same server, we can achieve an application-level session-persistent load balancing. Therefore, we propose to keep track of TLS sessions in the load balancer and use them to implement an application-level session-persistence load balancing. We maintain a *session table* in the load balancer to store the TLS session IDs issued by each server. Supporting

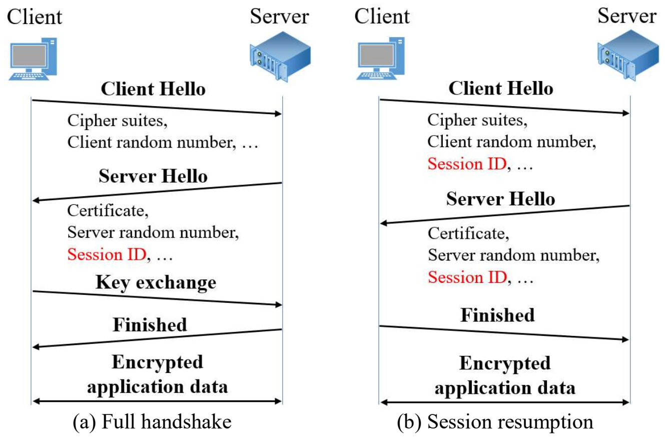


Fig. 2. TLS 1.2 handshake protocol

both session persistence and connection persistence means that we also need a connection table in the load balancer because it is impossible to forward different connections of a session (which have different 5-tuples) to the same server by hashing.

In our load balancing scheme, all incoming and outgoing packets pass through the load balancer. Algorithm 1 represents `the procedure carried out for incoming packets. The load balancer terminates each incoming TCP connection request by sending a SYN/ACK packet to the corresponding client. After establishing a TCP connection between a client and the load balancer, the client sends a *TLS_Client_Hello* message. The load balancer looks up the TLS session ID of the message in its TLS session table. If the message does not contain a session ID known to the load balancer, which means a new TLS session must be established between the client and one of the servers, the load balancer randomly selects one of the servers and transfers its endpoint of the connection along with the *TLS_Client_Hello* message to the server using a TCP hand-off protocol [14]. The load balancer also adds the 5-tuple of the connection to the connection table so the subsequent packets of the connection can be easily directed to the same server.

In response to the *TLS_Client_Hello* message, the server sends a *TLS_Server_Hello* message containing a session ID. The load balancer continuously monitors outgoing packets, and if it observes a *TLS_Server_Hello* message, it adds the included TLS session ID into the session table. The subsequent connection requests of the client will include the session ID, and the load balancer can easily direct them to the same server using the session table.

TLS 1.2 has also another resumption mechanism using tickets. In this mechanism, the server does not cache session secrets. Instead, it puts an encrypted form of the secrets into a ticket and passes it to the client. The client includes the ticket in its subsequent *TLS_Client_Hello* messages. The difference for us is that the server sends a ticket in a distinct message (*TLS_New_Session_Ticket*) during the handshake protocol. Moreover, the size of a ticket is typically larger than a session ID size. Hence, it is better to store a hash of tickets in the load balancer's session table. TLS 1.3 [15], which was published in 2018, has combined the two resumption mechanisms into one mechanism named Pre-Shared Key (PSK). It is very similar to the ticket mechanism. Its most important difference is that the server sends the ticket after finishing the handshake protocol, and consequently, the ticket message will be completely encrypted and unclear to the load balancer. To resolve this problem, we modified the web server program so that it sends a clear copy of the issued TLS tickets to the load balancer.

Algorithm 1. Procedure of handling client-to-server packets

```
Input: Received packet from a client: pkt
1:  if (pkt is TCP_CONNECTION_REQUEST) then
2:      Send SYN/ACK packet to the client
3:  else
4:      if (pkt is TLS_CLIENT_HELLO) then
5:          server ← LookUp(SessionID(pkt), sessionTable)
6:          if (server = NULL) then
7:              server ← SelectServer()
8:          end if
9:          TCPHandOff(pkt, server)
10:         AddEntry(FiveTuple(pkt), server, connectionTable)
11:     else
12:         server ← LookUp(FiveTuple(pkt), connectionTable)
13:         if (server = NULL) then
14:             drop(pkt)
15:         else
16:             forward pkt to server
17:         end if
18:     end if
19: end if
```

IV. Evaluation

To evaluate the proposed load balancing scheme, first, we prepared seven Linux virtual machines in a host system featuring 32 GB of RAM and an Intel Xeon E5-2683 processor. One of the VMs plays the role of a load balancer, another one hosts a database, and the other five VMs are web servers. We implemented our proposed load balancer in the VM dedicated to load balancing. To compare the proposed method with other approaches, we implemented Cheetah (as a layer-4 load balancer) and also installed Nginx and HAProxy (as layer-7 load balancers) on the VM.

Our simple web application makes use of application-layer sessions, i.e. it retrieves session data for each request. We designed the application so that it retrieves some data as session data for each application request. To evaluate different workloads, we can change the size of session data. In layer-4 load balancing, session data is stored in the database, but in layer-7 load balancing and also in our method, session data of each application session is stored in a web server.

There are various web server benchmarking tools such as Siege [16] and ApacheBench (ab) [17] which simulate many clients sending web requests, but a problem with all of existing tools is that they cannot associate multiple requests of each simulated client to 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. Hence, we modified the source code of Siege to have this feature. The modified Siege saves the TLS session ID assigned by a server for each client and uses the session ID for all subsequent requests of the client.

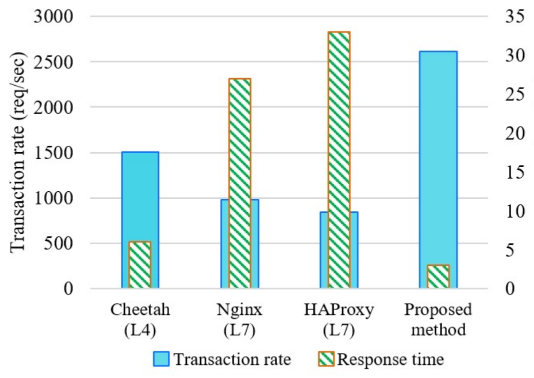


Fig. 3. Performance evaluation of the load balancing methods

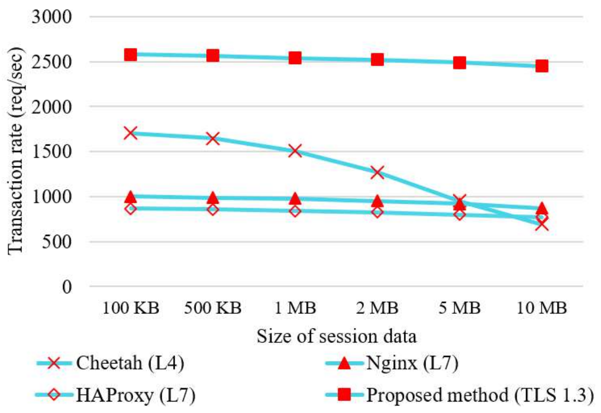


Fig. 4. Performance of load balancers for different sizes of session data

Finally, we evaluated the performance of all the load balancers in our testbed using the modified Siege. Fig. 3 shows the maximum request rate handled by each load balancer along with the average response time of each one. Cheetah, which is a layer-4 load balancing scheme, has a higher performance than Nginx and HAProxy because it does not act as a reverse-proxy. However, in layer-4 load balancing, the request processing capability of web servers becomes limited by the process of retrieving application-layer session data for each request. Therefore, the proposed method has achieved a significantly higher request processing rate than Cheetah. Moreover, the response time of the proposed method is considerably lower than the other load balancers. The results also show that our method works slightly better with TLS 1.2 compared to TLS 1.3.

In the previous experiment, the session data size was 1 MB. To evaluate different workloads, we repeated the experiment using different sizes of session data. Fig. 4 illustrates the transaction rate of the load balancers for different sizes of session data. While increasing the session data size does not considerably affect the performance of session-persistent LBs, it significantly degrades the performance of Cheetah.

V. Discussion

The first issue regarding the proposed method is that it only works on web servers that employ TLS. However, this requirement does not put an obstacle in the way of the proposed method because nowadays most of the web-based services use TLS, and especially, TLS is critical for the applications that need to recognize clients using sessions. 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 [13] has recommended an upper limit of one day for the session resumption lifetime while TLS 1.3 [15] has recommended an upper limit of seven days. It is also noteworthy that *TLS_Client/Server_Hello* messages are small (smaller than 200 bytes), and we do not need to be concerned about IP fragmentation or multiple TCP segments. The final issue is the problem of deploying the proposed load balancer in a distributed manner. Similar to Ananta, since the load balancer maps packets to servers only using tables, a change in the pool of load balancers can result in directing some packets to wrong servers. This problem can be resolved by employing the Beamer's method, i.e. daisy chaining. Further work on this is deferred to future research.

VI. Conclusion

In this paper, we presented a new approach to session-persistent load balancing for clustered web servers. The proposed method does not process packets up to the application level. Instead, it makes use of the TLS session resumption mechanism to achieve application-level session affinity. Hence, the proposed method can be categorized as a layer-4 load balancer. Our preliminary evaluation showed that the performance of the proposed method is significantly higher than the previous approaches in terms of transaction rate and response time. Our future work will focus on developing a distributed version of the proposed load balancer. We are also going to evaluate a realistic prototype of the load balancer in a real and high-demand web-server cluster and compare its results with the state-of-the-art L4 and L7 load balancers.

References

[1] D. E. Eisenbud *et al.*, "Maglev: A fast and reliable software network load balancer," in *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 523–535.
[2] V. Cardellini, and M. Colajanni, "Dynamic load balancing on web-server systems," *IEEE Internet computing*, vol. 3, no. 3, pp. 28–39, 1999.
[3] V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu, "The state of the art in locally distributed Web-server systems," *ACM Computing Surveys (CSUR)*, vol. 34, no. 2, pp. 263–311, 2002.
[4] P. Patel *et al.*, "Ananta: Cloud scale load balancing," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 207–218, 2013.
[5] C. Hopps, "Analysis of an equal-cost multi-path algorithm, RFC 2992," 2000.
[6] 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*, 2015, pp. 1–13.
[7] 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*, 1997, pp. 654–663.
[8] 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 18)*, 2018, pp. 125–139.
[9] T. Barbette *et al.*, "A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency," in *17th USENIX$ Symposium on Networked Systems Design and Implementation*, 2020, pp. 667–683.
[10] "Nginx." [Online]. Available: www.nginx.org.
[11] "HAProxy." [Online]. Available: www.haproxy.org.
[12] 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*, 2016, pp. 1–16.
[13] T. Dierks and E. Rescorla, "The transport layer security (TLS) protocol version 1.2, RFC5246," 2008. [Online]. Available: https://tools.ietf.org/html/rfc5246.
[14] K. Gilly, C. Juiz, and R. Puigjaner, "An up-to-date survey in web load balancing," *World Wide Web*, vol. 14, no. 2, pp. 105–131, 2011.
[15] E. Rescorla, "The transport layer security (TLS) protocol version 1.3, RFC8446," 2018. [Online]. Available: https://tools.ietf.org/html/rfc8446.
[16] "Siege." [Online]. Available: www.github.com/JoeDog/siege/.
[17] "ApacheBench." [Online]. Available: httpd.apache.org/.