Artemis: A Practical Low-latency Naming and Routing System

Xuebing Li[1,2], Bingyang Liu[3], Yang Chen[1,2], Yu Xiao[4], Jiaxin Tang[1,2], Xin Wang[1,2]
[1] School of Computer Science, Fudan University, Shanghai, China
[2] Shanghai Key Lab of Intelligent Information Processing, Fudan University, Shanghai, China
[3] Huawei Technologies Co. Ltd., Beijing, China
[4] Department of Communications and Networking, Aalto University, Finland
{chenyang,xinw}@fudan.edu.cn,liubingyang@huawei.com,yu.xiao@aalto.fi

ABSTRACT

Today, Internet service deployment is typically implemented with server replication at multiple locations for the purpose of load balancing, failure tolerance, and user experience optimization. Domain name system (DNS) is responsible for translating human-readable domain names into network-routable IP addresses. When multiple replicas exist, upon the arrival of a query, DNS selects one replica and responds with its IP address. Thus, the delay caused by the process of DNS query including the selection of replica is part of the connection setup latency.

In this paper, we proposed Artemis, a practical low-latency naming and routing system that aims at reducing the connection setup latency by eliminating the DNS query latency while keeping the ability to perform optimal server (replica) selection based on user-defined rules. Artemis achieves these goals by integrating name resolution into the transport layer handshake. Artemis allows clients to calculate locally the IP address of a Service Dispatcher, which serves as a proxy of hosting servers. Service Dispatchers forward the handshake request from a client to a server, and the response is embedded with the server's IP address back to the client. This enables clients to connect directly with servers afterward without querying DNS servers, and therefore eliminates the DNS query latency. Meanwhile, Artemis supports user-defined replica selection policies. We have implemented Artemis and evaluated its performance using the PlanetLab testbed and RIPE Atlas probes. Our results show that Artemis reduces the connection setup latency by 26.2% on average compared with the state-of-the-art.

CCS CONCEPTS

• **Networks** → **Network services**; **Naming and addressing**.

KEYWORDS

QUIC, naming service, network latency, anycast

ACM Reference Format:
Xuebing Li[1,2], Bingyang Liu[3], Yang Chen[1,2], Yu Xiao[4], Jiaxin Tang[1,2], Xin Wang[1,2]. 2019. Artemis: A Practical Low-latency Naming and Routing System. In *48th International Conference on Parallel Processing (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3337821.3337897

ICPP 2019, August 5–8, 2019, Kyoto, Japan

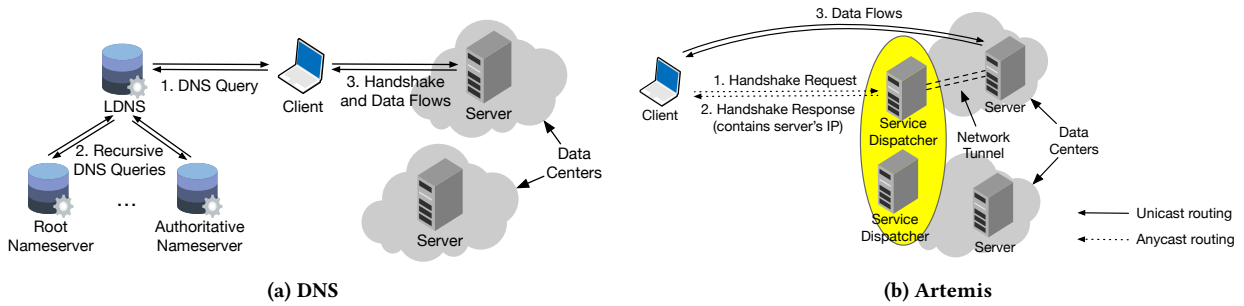
ACM ISBN 978-1-4503-6295-5/19/08…$15.00
https://doi.org/10.1145/3337821.3337897

1 INTRODUCTION

Low latency is a critical requirement for today's Internet services [1, 2]. Amazon found that a reduction of 100ms in page load time (PLT) contributes to an increment of 1% in revenue [2]. Similarly, Google reported that a 2s delay might cause a 4.3% loss in revenue per visit [3]. To reduce network latency, servers are expected to be deployed as close as possible to the users. In the scenarios where a set of replica servers sharing the same domain name are available at multiple locations, it is essential to take into account the difference in the latency between the client in question and each replica, when deciding which replica to select for serving a request.

DNS is responsible for mapping human-readable domain names into network-routable IP addresses. As illustrated in Fig. 1a, the process of setting up a connection between a client and a replica server consists of three steps. Firstly, a client sends a query for the IP address of a domain to a local DNS (LDNS) or a public DNS server, such as Google Public DNS and OpenDNS. Secondly, the DNS server makes recursive DNS queries to the nameservers. In the case of server replication, the client will receive a response which contains the IP address of a replica. Thirdly, the client connects to the replica based on the DNS response. The latency generated in the first two steps is called DNS query latency, while that in the third step is called handshake latency. The combination of them forms the connection setup latency of each request. To accelerate subsequent queries for the same domain name, the results are usually cached within the resolvers during the duration of time-to-live (TTL). Depending on many factors, e.g., cache hit rate, the DNS query latency typically ranges from 1ms to 5s [4], which can consume up to 13% of page load time (PLT) [5]. In this paper, we aim at reducing the connection setup latency, especially the DNS query latency, by redesigning the name resolution service while supporting replica selection based on user-defined policies.

To achieve the above-mentioned goals, we propose Artemis, a novel low-latency naming and routing system, which implements the process of connection setup in Fig. 1b. Our key idea is to perform name resolution as part of the handshake. An overlay layer composed by *Service Dispatcher*s is introduced. A Service Dispatcher works as a proxy for hosting servers of a group of web services. It accepts the client's handshake requests and forwards the requests to a replica server which is selected by the user-defined policies. A Service Dispatcher exposes its services by the *Service Address*, which is an anycast IP address composed of a predefined network prefix and a network suffix calculated by the domain name. The server embeds its IP address inside the handshake response so that



(a) DNS (b) Artemis

Figure 1: Connection setup process of DNS (on the left) and Artemis (on the right). DNS performs name resolution before connecting to the server. Artemis performs name resolution on completion of the connection handshake.

the client can send subsequent packets directly to the server, bypassing the Service Dispatcher. We call this process of migrating the connection from Service Address to the server's IP address as late binding. In this way, domain name resolution is performed without querying name servers. In this paper, we present our design based on the QUIC protocol[1]. We utilize QUIC's connection migration mechanism in the implementation of late binding.

We implemented Artemis and deployed it on a commercial anycast cloud called Packet[2]. Packet is available at four locations with anycast support around the world. The experiment result verifies the functionality of the system, showing that Artemis can be easily deployed on the Internet. We also compared the connection setup latency between Artemis and the state-of-the-art DNS-based solutions, using the PlanetLab testbed and RIPE Atlas probes. The results show that Artemis reduces the connection setup latency by 26.2% on average.

The contributions of this paper are summarized below.

- We designed and implemented a novel naming and routing system, Artemis. It eliminates the DNS query latency and significantly reduces the overall connection setup latency.
- The Service Dispatcher performs server selection based on user-defined policies. Our customized policy directs the client to the replica server of the lowest latency.
- The late binding module allows the server to notify the client of its IP address after finishing handshake. The server migrates to the new address without any additional latency.
- We deployed the Artemis prototype over a commercial anycast cloud. Experiment results have proven its scalability over the Internet. Our measurement shows that Artemis outperforms DNS regarding connection setup latency with the reduction of 26.2%.

The following of this paper is organized as below. Section 2 gives an overview of different server selection strategies. Section 3 and Section 4 describe the system design and implementation, respectively. The system evaluation is presented in Section 5. A brief discussion is given in Section 6 before we conclude our work in Section 7.

[1]QUIC is a newly proposed transport layer protocol by Google. It has been supported by various web service providers including Akamai and Cloudflare.
[2]https://www.packet.com/

2 RELATED WORK

The main functionality of Artemis is twofold. It maps domain names to IP addresses and performs (replica) server selection. The first functionality has been described in the previous section. In this section, we review the previous works related to three categories of server selection strategies. A comparison between Artemis and state-of-the-art solutions is shown in Table 1.

2.1 Anycast-based Solutions

Anycast addressing is a one-to-many association where packets are routed to any single member of a group of end hosts that are all identified by the same IP address. It utilizes the de-facto standard inter-domain routing protocol on the Internet (BGP [9]) to select the shortest of multiple routes to reach a destination IP address. In most cases, the selected end host is close to the client, whereas in some cases the destinations are located far away, e.g. thousands of kilometers [10], from the client.

Anycast-based server selection is used by DNS servers [11] and modern content delivery networks (CDNs) including Microsoft Azure [12] and CloudFlare [7]. Replicas are assigned with the same anycast address, which means the selection of replica is up to the anycast routing mechanism in use. Cloudflare [7], for example, is built purely on anycast. A client connects to a replica server directly with an anycast address. The advantage is the simplicity but the disadvantage is the unawareness of server load. Anycast itself does not support custom policies on server selection. Fastroute [6] is proposed as a load-aware anycast routing protocol. The servers are deployed following a tree structure consists of several layers. When nodes at lower layers (starting from leaf nodes) become heavily loaded, they will forward the incoming connections to the nodes at higher layers. The advantage is that each node only needs to maintain the information of its higher layer, instead of the whole system, reducing the complexity of the nodes. The disadvantage is that the redirection relies on DNS which incurs DNS query latency. Each DNS redirection costs at least one additional DNS query.

2.2 DNS-based Solutions

In DNS-based solutions, an authoritative nameserver maintains a global view of the Internet and determines the best server for a given client based on user-defined policies, including network latency, server load, and so on. Akamai [8] utilizes such technology

System	Methology	Server Load Awareness	Connection Setup Latency	Server Selection Result
Fastroute [6]	anycast & DNS	Yes	high, with DNS latency	suboptimal
Cloudflare's work [7]	anycast	No	low	suboptimal
Akamai's work [8]	DNS	Yes	high, with DNS latency	optimal
Artemis	anycast	Yes	low	optimal

Table 1: Comparison of related works on server selection.

in its CDN network to achieve low latency. Google employs similar technology in its fronted serving architecture [13]. The advantage is that the nameserver has an overview of the system and the network. It can select any replica for the client and can take application-level information into account. However, the disadvantage is that the DNS query brings additional latency to the connection setup process.

Our work takes advantage of both anycast's fast connection setup and DNS's flexible controllability, resulting in a system that supports low-latency connection setup while performing optimal server selection based on user-defined policies.

2.3 Application-based Solutions

Some services reselect server after the client has connected to one of the servers. For example, HTTP status code 3xx for redirection is commonly used in web services [14]. Application-based server selection takes into account the status of the client. However, the redirection takes up much latency, making it not suitable for latency-sensible web applications.

3 DESIGN

In this section, we give an overview of the whole system in Section 3.1 and describe each module in detail from Section 3.2 to 3.5.

3.1 Overview

The structure of Artemis is shown in Fig. 2. In the architecture of Artemis, we remove conventional DNS servers and introduce a new element called Service Dispatcher. Service Dispatchers are deployed in data centers and serve as a proxy, through which the handshake messages between clients and servers are exchanged.

Each Replica Server has a Service Address, which is an anycast address corresponding to its domain name. These addresses are calculated by the ServiceID mapping module as described in Section 3.2. But the replica server does not advocate the Service Address by itself. Instead, it relies on a Service Dispatcher, which advocates the Service Address and forwards the received packets to it. One Service Dispatcher can be connected to a bunch of replica servers of different domain names. The anycast network prefix is advocated via BGP announcements from all of the data centers. Service Dispatchers located in different data centers are fully connected with Inter-DC tunnels to transfer the client's handshake packets across zones without any modification. There are a variety of protocols to implement the tunnels and we choose Generic Routing Encapsulation (GRE) in our implementation.

The process of connection setup is described below. Firstly, a client calculates Service Address according to the domain name

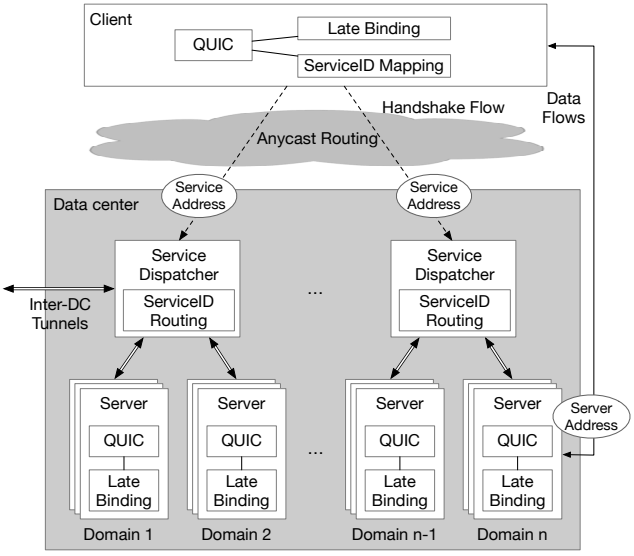


Figure 2: Artemis architecture within a data center. A client connects to a Service Dispatcher through the Service Address. The Service Dispatcher forwards the handshake request to the optimal replica server. The replica server migrates its IP address from the Service Address to its uincast IP address (the server address in the diagram) for the connection.

(described in Section 3.2) and sends a handshake request to the calculated address. Upon the arrival of the request via anycast routing, the Service Dispatcher in question selects one data center based on user-defined rules and forwards request to the Service Dispatcher in that data center. Ideally, the request should be forwarded to the nearest data center with service deployment, and the Service Dispatcher there will choose a replica server for the domain in the data center and forward the request to it (described in Section 3.3 and 3.4). The replica server processes the request and sends back a handshake response with its unicast IP address embedded. The response is sent to the client through a Service Dispatcher in the same data center, and the handshake process completes when the client receives the response. After that, the client migrates the connection with Service Address to that with the replica server's unicast IP address. This process called late binding is described in Section 3.5.

3.2 ServiceID Mapping

The ServiceID mapping module resides in the clients and the Address Managers. The Address Managers are deployed inside every

data center and are responsible for allocating Service Addresses to the replica servers. The ServiceID mapping module takes domain names as input and generates Service Addresses as output. The generated Service Address is composed of two parts: a service prefix and a ServiceID.

The service prefix is the same as subnet ID in IP addresses and is advocated by BGP routers within all of the data centers, making it an anycast subnet. If a client wants to use Artemis, it needs to acquire the network prefix from the service provider at first because the service prefix is used to uniquely identify an instance of Artemis. Anycast routing provides a simple but efficient solution to direct a client to its nearest data center. Artemis takes advantage of this feature as previous works described in Section 2.1.

The ServiceID is generated by the domain name with a predefined hash function. The domain name is hashed with SHA512 [15], and the first N_{sid} bits are chosen to be the ServiceID, where N_{sid} is the length of the ServiceID in bits. N_{sid} is determined by the cloud provider's requirements. Generally, the longer the ServiceID is, the more allocable addresses there are. However, it also means more costs to be spent on acquiring IP addresses. A good hash function maps the Service Addresses as evenly as possible over the Service Addresses. Ideally, the Service Dispatchers are under similar loads.

With the design of Service Address, we manage to achieve three goals. Firstly, the destination IP address is derived from the domain name without any network-based queries (E.g., DNS query). Secondly, the Service Dispatcher with the calculated service address is highly likely to be located in a nearby data center, following the principle of geolocation based anycast routing. Thirdly, ingress traffics are distributed among all of the Service Addresses, which means we can simply increase N_{sid} to raise the number of Service Dispatchers, reducing the load of each Service Dispatcher.

3.3 End-User Mapping

The end-user mapping module implements user-defined replica server selection policies. Our design aims at finding a replica server with the lowest latency to the client in question since low latency is a common requirement of web services [16]. Note that Artemis can support other customized policies as well.

From the handshake request, we can find out where the request comes from (the client source address) and what service the client requests for (the domain name)[3]. The end-user mapping module takes the client IP address and the domain name as input and returns the closest data center that contains at least one replica server as output.

We use a database to provide data storage of high scalability and reliability. There are two tables named *ServiceDeployment* and *LatencyMeasurement*. Just as the name indicates, ServiceDeployment records the deployment status of different domain names across data centers, while LatencyMeasurement records the network latency between the clients and the data centers.

Whenever a new replica server is deployed, a new record (domain name, data center) is inserted into the ServiceDeployment table. On the contrary, when all of the replica servers of a domain have been removed from a data center, the record (domain name, data center) is removed from the ServiceDeployment table. Regarding the LatencyMeasurement table, we perform active RTT measurements between every pair of data center and client subnet periodically. A record (subnet, data center, RTT) is updated in the LatencyMeasurement table when the measurement completes.

The replica server selection procedure is as follow. Given a client subnet and a target domain name, we query the ServiceDeployment table at first to find out the data centers where replica servers with the given domain name exit. Then, based on the previous result set, we query the LatencyMeasurement table to find out the nearest data center to the client.

The database is deployed in the master-slave mode to provide high accessibility. The master node handles write operations, and the slave nodes handle read operations. Slave nodes are deployed in every data center so that Service Dispatchers can perform queries quickly.

3.4 ServiceID Routing

The ServiceID Routing module is designed to forward packets to the optimal replica server for each client. It is composed of two parts, the inter-DC routing, which directs packets to the optimal data center, and the intra-DC routing, which directs packets to a proper server within the selected data center. The packet path is shown in Fig. 3.

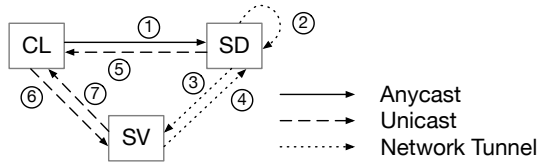
3.4.1 Inter-DC Routing. At first, the client sends a handshake request to the Service Address (step 1). Every Service Address has its corresponding Service Dispatcher inside every data center. The request is possible to reach the Service Dispatcher inside any of the data centers. The inter-DC routing infrastructure is designed to route the handshake request from where it is received, $DC_{anycast}$, to the optimal data center $DC_{optimal}$, which is determined by the end user mapping module.

On receiving the handshake request, the Service Dispatcher extracts the domain name from the request and uses the end-user mapping module to calculate $DC_{optimal}$. If $DC_{optimal}$ refers to a different data center, the handshake request is forwarded through the network tunnels to $DC_{optimal}$ (step 2). After reaching $DC_{optimal}$, intra-DC routing is conducted to forward the request further to a replica server.

3.4.2 Intra-DC Routing. The next step is to dispatch the handshake request from the Service Dispatcher to the corresponding replica server within the data center. A database table called ReplicaDeployment is created to record the information of all replica servers. Whenever a new server is registered to a domain, a new entry (domain, tunnel name) is inserted into the table. The entry is deleted when the server stops to serve for the domain.

Upon receiving the handshake request, the Service Dispatcher queries the ReplicaDeployment table and selects a replica server for the domain. We perform random selection among server replications, but more sophisticated rules can be supported. The Service Dispatcher sends the handshake request to the selected replica server through a network tunnel (step 3). Afterward, the replica server receives the original handshake request from the client. After processing, the replica server generates a handshake response and sends it back to the client. To avoid being blocked by the client's

[3]QUIC enforces the handshake request to contain the server's domain name, as the value of an extension in TLS1.3 called server name indication (SNI) [17].



No.	Src	Dst	No.	Src	Dst
1	CL.IP	SA	5	SA	CL.IP
2	CL.IP	SA	6	CL.IP	SV.IP
3	CL.IP	SA	7	SV.IP	CL.IP
4	SA	CL.IP			

Figure 3: Packet paths of handshake flow and data flow in Artemis. The table shows various packet headers. Symbols in block letters represent different roles in Artemis. CL is the client, SD is the Service Dispatcher, and SV is the server. SA stands for Service Address and is the IP address of the Service Dispatcher. CL.IP is the IP address of the client. It may be the router's IP address if the client is behind a NAT. SV.IP is the unicast IP address of the server. The network tunnels transfer packets between two hosts bidirectionally.

Condition		Action
in_port	optimal_check	output
N_{bu}	true	TB_{si}
N_{bu}	false	TB_{bi}
TB_{si}	/	N_{bu}
TB_{bi}	/	TB_{si}

Table 2: The routing table of a Service Dispatcher.

firewalls, the response packet is delivered back to the client through the original route (step 4 and 5). From the client's view, it has established a connection with the Service Dispatcher, which is actually a proxy for the replica server. Then, the connection is migrated to the server's unicast IP address with the help of the late binding module. Finally, the client and the server can communicate directly with unicast IP addresses (step 6 and 7).

Putting all of the above together, the routing table of the Service Dispatcher is shown in Table 2. Denote network tunnels towards the other Service Dispatchers by TB_{bi}, and the ones to the replica servers by TB_{si}. N_{bu} refers to the port which is assigned with the Service Address in a Service Distributor. *optimal_check* indicates whether $DC_{anycast}$ equals to $DC_{optimal}$.

Artemis gains three advantages by the design of ServiceID routing. At first, the routing policy is aware of rich elements including the system state, the network performance, and the application-level information. Then, all of the data mentioned above are persisted in the database. The Service Dispatchers are stateless and can recover from failures quickly. Finally, network tunnels enable us to complete the routing by only modifying a small part of the replica servers' network stacks. More detail about the implementation is given in Section 4.1.

3.5 Late Binding Module

When initiating a connection, the client sends a handshake request to the Service Dispatcher. The handshake reaches the server with the help of the ServiceID routing. The handshake response follows the same path back to the client. The connection is established within one round trip time (RTT). But the Service Dispatcher is designed only for routing the initial packets. After the handshake, packets should be delivered directly between the client and the server. It requires the server to migrate the connection from Service Address to its own unicast IP address. The late binding module is introduced for such purpose.

QUIC already supports connection migration on completion of the handshake, called Server's Preferred Address [18]. The server is allowed to accept connections on one IP address and attempt to transfer these connections to a more preferred address shortly after the handshake. The mechanism is described below.

Both the handshake request and the handshake response are allowed to carry transport parameters. The transport parameters are defined as key-value pairs. The key is of integer format and the value is of binary format with flexible length. The server embeds its preferred address inside the handshake response with the key of preferred_address (0x000d). The client extracts the preferred address and sends further packets to the server's new address immediately. The late binding is fast because it happens at the same time of handshake and no additional RTT is required. Since QUIC has built support of mobility, the change of server's IP address does not complicate either the server's or the client's state machine.

We find that almost all of the QUIC-IETF implementations have supported the transport parameter *preferred_address*. But none of them have implemented the logic of migrating the connection[4]. To the best of our knowledge, we are the first to implement this functionality.

Besides, we propose the cache mechanism of Server's Preferred Address at the client's side. A key named service_ttl (0xff00) is registered as a new transport parameter. The value is in the format of integer, representing how many seconds that the Server's Preferred Address should be cached at the client's side. As long that the TTL is not expired, the client is allowed to make new connections to the service replica via the server's unicast IP address directly. Otherwise, the client must query for the optimal server again by connecting to the Service Address.

4 IMPLEMENTATION

We deployed Artemis in Packet, an Anycast cloud. At the time of writing, it provides hosts with anycast addresses in Amsterdam, New York, Tokyo, and California. In each data center, we ran a Service Dispatcher on one physical machine and chose another two machines as replica servers. All of them serve the same domain name. Section 4.1 presents the implementation of the ServiceID routing layer, while Section 4.2 presents our change to the network stack.

[4]We have checked ngtcp2, Minq, mozquiz, picoquic, and quicly.

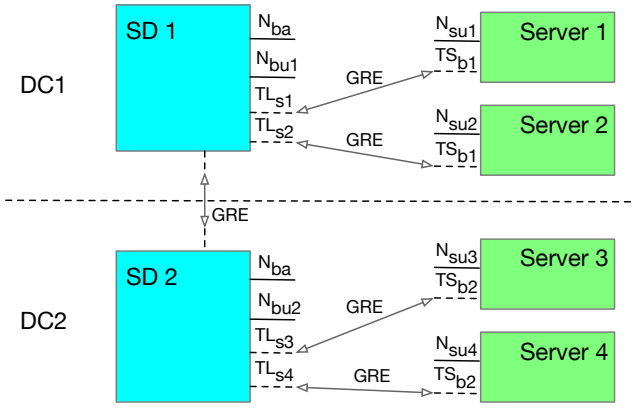


Figure 4: An example of Artemis deployment across two data centers. All of the machines serve the same domain. DC stands for data center and SD stands for Service Dispatcher. Solid lines represent physical NICs, dashed lines represent virtual interfaces, and lines with arrays represent GRE tunnels.

4.1 Tunnels and Packet Forwarding

We used GRE tunnels to perform layer 3 packet forwarding. It encapsulates raw IP packets inside a GRE packet and transfers the whole IP packet to the paired endpoint through an IP network. Open vSwitch (OVS) [19] was used in our system for tunnel establishment and packet forwarding.

Fig. 4 illustrates an example scenario of Artemis deployment across two data centers. A Service Dispatcher is equipped with two physical NICs, namely N_{ba} and N_{bu}. N_{ba} is assigned with one or more anycast IP addresses (ServiceIDs) for the anycast routing. N_{bu} is assigned with a unicast IP address for establishing GRE tunnels. The server is equipped with only one physical NIC, namely N_{su}. It is assigned with a unicast IP address for establishing GRE tunnels and performing data transmissions.

GRE tunnels and virtual interfaces are created on the Service Dispatchers and the replica servers. A Service Dispatcher establishes GRE tunnels to the Service Dispatchers in other data centers, namely TB_{bi}, and to the replica servers in the same data center, namely TB_{si}. For both of the Service Dispatchers and the servers, a virtual interface is created for each GRE tunnel to provide read and write access to the applications (I.e., QUIC servers). The OVS is configured to forward all packets from virtual interfaces to the other endpoint through GRE tunnels. The IP addresses of the Service Dispatchers' virtual interfaces are not necessary and are set randomly. This is because we can simply specify the remote peer by writing to specific virtual interfaces. Regarding the servers, TS_{bi} is the other endpoint of TB_{si}. It connects the Service Dispatcher (denoted as bi) and is configured as the same IP address as ServiceID, which is also the IP address of N_{bu} for bi.

The Service Dispatcher does not generate any extra packets, it simply routes received packets to their proper destinations. The routing policy is implemented in two parts. For the ingress packets arriving from N_{ba} and TB_{bi}, we wrote a program to parse and route QUIC handshake packets. It creates a socket for each interface, including physical NICs and virtual interfaces. The sockets were bound to the interfaces via the socket option SO_BINDTODEVICE. Then it used raw socket to receive IP packets from the interfaces and parsed the QUIC handshake packet. According to the routing policies described in Section.3.4, the packets were forwarded to specific destinations by writing to the corresponding socket directly. The socket option IP_HDRINCL was set to prevent the system from filling in the IP headers. For the ingress packets arriving from TB_{si}, we created data flows in OVS to send out all packets received from the servers (with in_port as TB_{si}) to the Internet via N_{ba}. Since TS_{bi} was already configured with the ServiceID, which was of the same address as N_{ba}, the Service Dispatcher do not need to change any fields of the IP packet. The packets were able to bypass the source address check.

4.2 Network Stack Changes

As mentioned in Section 3.5, we implemented QUIC's "preferred address" feature and the ServiceID Mapping module.

On the client side, we patched the DNS module with the ServiceID Mapping module. Linux uses the system call *getaddrinfo* to perform DNS query. We added a new system call *getaddrinfosid* which accepts the same input as *getaddrinfo* but returns the IP address calculated by the algorithm described in Section 3.2. A QUIC client can change from DNS to ServiceID mapping by simply changing *getaddrinfo* to *getaddrinfosid*. After the completion the handshake, the client changes the IP address of the remote peer by creating a new socket to replace the old one.

On the server side, we modified the QUIC server to make it work on multiple interfaces with a finer control over the routing. The legacy QUIC server creates only one socket and listens on a UDP port 443. Upon receiving a packet, it extracts the packet source address and sets it as the IP address of the remote peer. This mechanism works on multiple interfaces by leaving the choice of the outgoing interface to the system routing table. We created a socket for each interface, including the physical NIC (N_{su}) and the GRE tunnels (TS_{bi}). The socket and the interface are bound with the socket option SO_BINDTODEVICE. Upon receiving a packet, the QUIC server records which socket it reads from and performs further write operations on the same socket. So that the responses can follow the same path as the request back to the client. The late binding module utilizes these features by changing the socket from TS_{bi} to N_{su} when the client changes the remote peer from the Service Address to the server's IP address.

Our modification to the QUIC client and the QUIC server is based on ngtcp2[5]. The modification takes about only 50 lines of code. We did not change anything to the QUIC protocol.

5 EVALUATION

In order to validate the usefulness of the proposed system, we provide component-level analytics over the system in Section 5.1. To reveal the benefits gained by Artemis with a large scale deployment, we evaluate it based on the real-world Internet measurements in Section 5.2.

[5]https://github.com/ngtcp2/ngtcp2

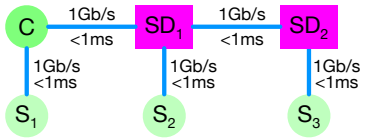


Figure 5: Network topology used for benchmark: client (C), server (S), and Service Dispatcher (SD).

5.1 Component-Level Performance

In this section, we present a performance analysis of the system based on micro-benchmarks. We deploy the network shown in Fig. 5. Each host is a 4-core (8-thread) 3.40 GHz Intel Xeon running Linux Ubuntu-16.04 (64-bit). In our experiment, a client sends concurrent queries to a server. For each query, the client connects to a server and sends a request after handshake. Then, the server responds with a plain text "Hello World". The benchmark network simulates three scenarios: (1) The client connects to the server with IP address ($C \rightarrow S_1$), (2) the client connects to the server via Artemis when anycast chooses the optimal data center ($C \rightarrow SD_1 \rightarrow S_2$), and (3) the client connects to the server via Artemis when anycast chooses a sub-optimal data center ($C \rightarrow SD_1 \rightarrow SD_2 \rightarrow S_3$).

Metrics In each scenario, we measured three latency metrics:

- handshake latency, the time spent on establishing a QUIC connection with the IP address
- connection setup latency, the time spent on establishing a QUIC connection with the domain name
- query latency, the time spent on establishing a QUIC connection with the domain name and completing the "Hello World" query

In Artemis, the connection setup latency is the same as the handshake latency because Service Addresses are calculated locally. The calculation delay is negligible compared to network latencies. Besides, we recorded the CPU and memory usage on each host.

QUIC Client/Server Performance

Fig. 6 shows the result of scenario 1. Legacy QUIC client and server implementations were used in this scenario. We ran each test case five times to avoid stochastic errors and calculated the median value to represent repeated experiments. The results show that most of the metrics, including the handshake latency, the query latency, and the memory usage, increase linearly as the number of concurrent connection grows. The CPU usage keeps steady when the number of connections is larger than 40 on the client side. It is because I/O, instead of CPU, is the bottleneck at this time. The CPU usage reaches 30% when there are 1000 concurrent connections.

Overhead of Artemis

Fig. 7 shows the measurement results in scenario 2 and 3. On average, Artemis delays the handshake (query) process by 20.9 ms (27.3 ms). The delay is caused by Service Dispatchers because they need to forward clients' initial packets. Our implementation is in user space. It receives packets from the kernel space, parses packets in the user space, and delivery packets via the kernel space. This process can be accelerated if the program is running in kernel space, which is left for future work.

Unlike the clients and the servers, the memory usage of the Service Dispatchers keeps steady as the increase of concurrent connections. It is because Service Dispatchers are stateless. They do not need to maintain state machines for ongoing connections. The CPU usages of both clients and servers are almost the same in both cases. It proves that Artemis does not harm the client's and the server's performance.

Scalability

Artemis is of high scalability for two reasons. Firstly, it takes advantage of anycast, which provides a simple but efficient solution for directing requests from clients to their nearby replica servers. Unlike broadcast, which discoveries the best destination by sending duplicated requests to all of the candidates, no redundant packets are transferred for anycast. The reason is that BGP, the de-facto inter-domain routing protocol on the Internet, by nature selects the shortest of multiple routes to reach a destination IP address [9]. Therefore, our anycast-based mechanism is highly scalable.

Secondly, most of the components in Artemis are stateless. The Service Dispatcher's only job is to parse the first packet of a connection and to forward it to a proper replica server based on the query result from a stateful distributed database. It does not need to maintain the state of the connection because the server's response is routed directly to the Internet without special operations and the connection migration completes after the first round trip. Since there is no affinity between a Service Dispatcher and a replica server, each of them is easy to be replaced by a new one if it fails. All of the states are persisted in a distributed database built on MySQL. It uses the master-slave architecture which has already been proven to be easy to scale up [20].

5.2 Internet-Scale Performance

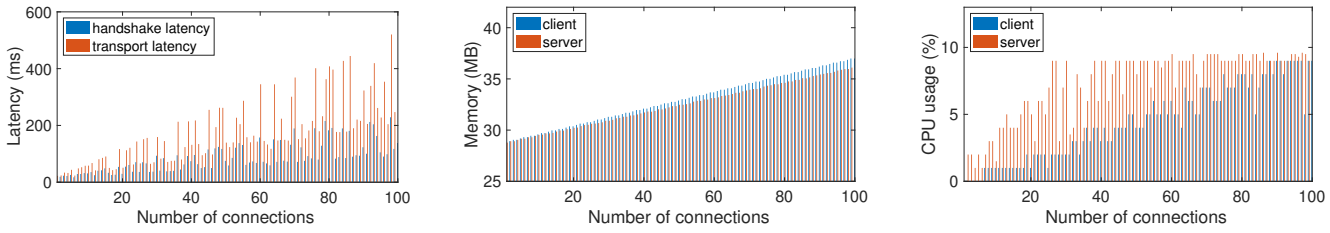
In this section, we simulate the performance of Artemis based on the benchmark network shown in the previous section and a real-world measurement of latency from the Internet.

Dataset

DNS Latency PlanetLab is a global research testbed that consists of servers in hundreds of institutions around the world. We used 80 PlanetLab nodes to measure the DNS latency. The clients use predefined DNS resolvers and the domain names are hosted by AWS Route53[6]. We focused on two metrics. 1) $latency_{hit}$, which is the DNS latency when a cache hit occurs, and 2) $latency_{miss}$, which is the DNS latency when a cache miss occurs.

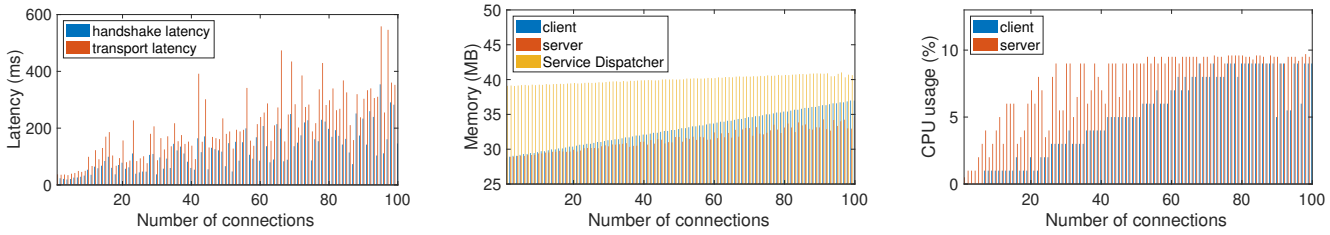
The result is illustrated in Fig. 8. The data shows a long-tail pattern for both types of latencies. When a cache hit occurs, the query latency is smaller than 2 ms in 68.4% of the cases. But there are also 7.0% of the cases where the latency is longer than 10 ms. In the case of cache miss, the latency grows and the tail expands further since the DNS resolver needs to traverse a long way to reach the authoritative name server. The result shows that 10.5% of the queries are facing latencies over 100 ms, which delays the connection setup latency severely.

[6] https://aws.amazon.com/route53/



(a) Latency (b) Memory (c) CPU Usage

Figure 6: Performance over various number of concurrent connections when the client connects to the server directly.



(a) Latency (b) Memory (c) CPU Usage

Figure 7: Performance over various number of concurrent connections when the client connects to the server via Artemis.

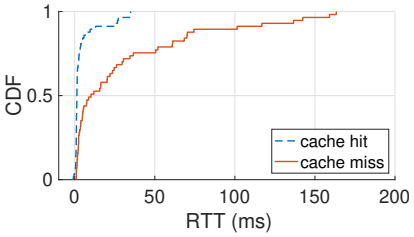


Figure 8: DNS latency on cache hit and cache miss.

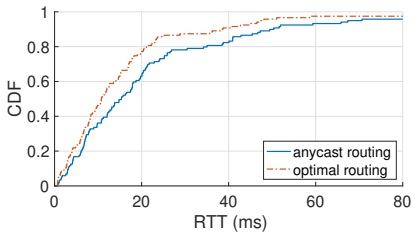


Figure 9: Connection latency by anycast routing and the optimal routing.

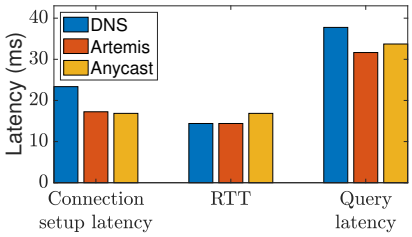


Figure 10: Comparison of latencies by DNS, Artemis, and anycast.

Anycast Routing Cloudflare is a leading CDN service provider. It provides service via 152 data centers around the world. Its public DNS servers are exposed via an anycast IP address 1.1.1.1 and are hosted globally. We measured the latency from residential IPs to the public DNS servers and analyzed the effectiveness of the commercial anycast network.

We used 500 RIPE Atlas probes as vantage points (VPs) to make DNS requests to public DNS servers. The requested domain is hosted on an authoritative name server owned by ourselves. The public DNS servers act as resolvers and forwards the query to our name server for DNS records, with a unicast IP address as the source address. In this way, we can reveal the unicast IP address of the DNS resolver which is chosen by anycast routing.

The unicast IP address and the anycast IP address of a DNS resolver do not reside in the same subnet. They may be advocated by different eBGP routers, resulting in different routing paths in the Internet. Therefore, a unicast address is representative for a DNS resolver if and only if it shares the same network path when reached from a source that is directed to the resolver via anycast. From each probe, we send traceroutes to both the anycast address and the unicast address of the DNS resolver. Then, we compare the IP address of the last hop of the paths. We find that in 322 of the records, the unicast address is representative for the DNS resolver. The other records are exempt from further analysis. Furthermore, we located the DNS resolvers by querying an IP geolocation database, GeoLite, with their unicast addresses. The DNS resolvers reside in 67 data centers, which is a subset of Cloudflare's deployment.

Knowing the unicast representative of each anycast DNS resolver, we can find out if anycast routing directs a VP to a DNS resolver of the minimal RTT. We used RIPE's ping utility to measure the RTT from each VP to each DNS resolver. $latency_{anycast}$ is the measured RTT from a VP to an anycast DNS resolver's unicast representative and $latency_{min}$ is the minimal RTT from a VP to all of the DNS resolvers. Among the results, 82.3% of the VPs have witnessed that $latency_{anycast}$ equals $latency_{min}$. It indicates that anycast routing already provides good performance in most cases. For the remaining VPs where the anycast routing does not lead to the optimal replica, we show the distribution of the RTTs in Fig. 9. Analyzing the additional latency $latency_{extra} = latency_{anycast} - latency_{min}$, we find that in 31.6% of the cases $latency_{extra}$ is smaller than 2

ms and in 15.8% of the cases, $latency_{extra}$ is larger than 20 ms. On average, anycast routing causes an additional latency of 12.71 ms compared to connecting to the nearest DNS resolver.

Simulation Results

We simulate empirical end-to-end network latencies modeled after latency patterns that have been observed in the above-mentioned measurements. We consider the scenario where a service is deployed in all data centers. The distribution of network link delays is shown in Table 3. We compare Artemis with two state-of-the-art techniques: DNS and anycast, which are widely used inside Akamai's [8] and Cloudflare's [7] CDN, respectively. Fastroute is based on anycast but relies on DNS redirection for load balancing, resulting in a longer connection setup latency than the other two. So we do not consider Fastroute in our simulation.

For DNS, the query delay is simulated according to the DNS latency dataset. Jung et al. conducted measurements in campus networks and found that the DNS cache hit rate was between 80% and 87% [21]. We choose the average value of 83.5% in our simulation. For anycast, the network delay from a client to a server follows the distribution that we have witnessed in anycast routing, as shown in Fig. 9.

For Artemis, the client's initial packet is routed by anycast routing and reaches $DC_{anycast}$ (SD_1). According to the measurement data, the data center is the optimal one for the client in 82.3% of the cases. In the other 17.7% of the cases, the packet is forwarded to $DC_{optimal}$ (SD_2). But we did not find a way to measure the RTT from $DC_{anycast}$ to $DC_{optimal}$ directly. Instead, we use the geographical distance to estimate the network latency: $latency_{overlay} = \frac{2.3 \times distance}{c}$. c is the speed of light and 2.3 is the median factor that network latency is slower than the speed of light [22]. Nowadays, ingress traffics go through load balancers to reach the servers. Service Dispatchers takes the role of load balancers in Artemis. Since inner-DC latencies are usually much smaller than inter-DC latencies, without loss of fairness, the latency between the load balancer and the servers are ignored for both Artemis and DNS. Since both DNS and Artemis find the optimal replica server for a client, packets between a client and a server always follow the optimal routing.

We set the concurrency to 1 and repeated the benchmark experiment with simulated network delays. The result is shown in Fig. 10.

Connection Setup Latency: On average, it takes 23.36 ms for a client to establish a new connection to a server via DNS. In Artemis, this process requires only 17.25 ms on average. In other words, the latency is reduced by 26.2% compared to DNS. The major reason for the reduction in latency is that Artemis saves the time spent on the name query. The ServiceID is generated locally, so a client can send out the initial packet based on the target domain name immediately. On the contrary, DNS requires a network-based query before connecting to the server. The only factor that may "delay" Artemis is the bad routing of anycast. In rare cases, the connection setup latency in Artemis can be as large as 200 ms. This is because anycast routing directs the client's initial packet to a distant replica server. Most anycast-based systems suffer from this problem [23, 24]. In Artemis, to solve the bad routing of anycast, we sacrifice a little in handshake latency but let a client select the nearest replica server. This is a worthy tradeoff because handshake latency occurs only once but a client can always benefit from the reduction in RTT during the life time of a connection. In our simulation, the connection setup latency of Artemis is 0.39 ms longer than the anycast solution. But it saves the RTT by 2.46 ms when compared to anycast. Besides, some solutions are proposed to improve anycast routing. For example, Li et al. added geographic hints to BGP advertisements and obtained encouraging results [10]. Artemis can benefit from them.

System	Network Link	Distribution of the Delay	Probability
DNS	$C \rightarrow DNS$	cache hit in Fig. 8	83.5%
		cache miss in Fig. 8	16.5%
	$C \rightarrow S_1$	optimal routing in Fig. 9	100%
Artemis	$C \rightarrow SD_1$	anycast routing in Fig. 9	100%
	$SD_1 \rightarrow S_2$	ignored	82.3%
	$SD_1 \rightarrow SD_2 \rightarrow S_3$	estimated by distance	17.7%
	$C \rightarrow S_2/S_3$	optimal routing in Fig. 9	100%
anycast	$C \rightarrow S_1$	anycast routing in Fig. 9	100%

Table 3: Distribution of network link delays.

RTT and Query Latency: The query latency is composed of the connection setup latency and one RTT between the client and the replica server. Since both DNS and Artemis can find the optimal replica server for a client, the RTT is the same for both of them. Anycast obtains longer RTT because anycast routing does not guarantee to find the nearest replica server. The client completes the query within 37.76 ms, 31.65 ms, and 33.72 ms for DNS, Artemis, and anycast, respectively. Artemis achieves the lowest latency by performing name query locally and fixing the bad routing caused by anycast. On the one hand, it is load-aware compared to anycast. On the other hand, it reduces the query latency by 16.2% compared to DNS.

6 DISCUSSION

In this section, we discuss related topics that are not covered in the previous sections.

6.1 Security

One security issue that comes with DNS's hierarchy architecture is the man in the middle (MITM) attack. For example, a LDNS, ordinarily owned by an ISP, can break or even imitate a service by returning the IP address of a malicious server to the client. Artemis avoids this problem by handling name resolutions purely on the server side. All of the Service Dispatchers are deployed and managed by the Artemis provider, e.g., a cloud provider like Amazon. The Service Dispatcher is guaranteed to never direct a client's request to a wrong replica server maliciously.

Another security issue in DNS is that an LDNS can easily track the browsing history of a client since all of the client's DNS queries must go through it. Artemis protects the client's privacy in twofold. At first, as mentioned above, all of the components in Artemis, except the clients, are managed by the Artemis provider. The user's information is not possible to be leaked to any third party. Secondly, a client can a) identify a replica server by its SSL certificate to avoid

a malicious server from mimicking a domain name and b) encrypt messages to prevent potential monitoring behaviors by middlewares like the Service Dispatcher. In fact, both of these security options are mandatory in QUIC and TLS1.3 [17, 18].

6.2 Deployment Requirements

Although Artemis is a novel infrastructure for naming and routing, we have tried our best to keep it compatible with today's Internet architecture. It is deployable in data centers without any change on the existing network infrastructures. The Service Dispatchers are hosted as dedicated servers and reside in the anycast address space. The anycast address space is isolated from the unicast address space of the servers, so they can be deployed incrementally on the cloud.

Our change on the client's and server's network stack obeys the design of QUIC. Others can refer to our implementation when utilizing QUIC's "preferred address" feature. The ServiceID mapping module is implemented as a patch to the client's DNS module. A client program can switch between DNS and Artemis by simply enabling/disabling the ServiceID mapping patch. Since we did not change the protocol, existing applications running over QUIC can switch to Artemis without any modification.

Both Artemis and DNS reveal the server's unicast address for a client. But Artemis is not meant to replace DNS. Instead, it provides an alternative naming solution that coexists with DNS. Artemis is more suitable for the scenario where replica servers are deployed globally. One potential scenario is the cloud-based services. Cloud providers register anycast network prefixes, distribute domain names to the customers and deploy the Artemis infrastructure. Customers deploy replica servers directly in the selected cloud's data centers. End users are able to obtain low-latency services by applying the ServiceID mapping patch provided by the cloud. Besides, a client is allowed to send name resolution requests to DNS and Artemis in parallel and uses whichever results received first from any of them.

7 CONCLUSION

In this paper, we propose Artemis, a low-latency naming and routing system that reduces the connection setup latency by eliminating the DNS query latency. With an additional layer composed of Service Dispatchers, Artemis can intervene in the handshake process and achieve optimal server selection based on user-defined policies for the client. The evaluation shows that Artemis is able to reduce the average connection setup latency by 26.2%, compared to the state-of-the-art DNS solution. Artemis represents an exciting new opportunity to reduce DNS query latency and provides web services with lower latency.

In the future, we plan to add the support of TCP to Artemis. Besides, we are going to implement Service Dispatchers purely with OVS and only use network-layer information when dispatching packets (without parsing the hostname indication from the QUIC packets). We will design a conflict list which records all of the domains whose Service Addresses fall into the same value. These domains are assigned with special Service Addresses, instead of the hash-generated one. A client must look up the list before calculating the Service Address. In this way, we can ensure the one-to-one mapping between a domain and a Service Address.

ACKNOWLEDGEMENT

We would like to thank our shepherds, Masaaki Kondo and Aaron Smith, as well as the reviewers for their helpful comments. Bingyang Liu and Yang Chen are the corresponding authors.

REFERENCES

[1] B. Briscoe, A. Brunstrom, A. Petlund, D. Hayes, D. Ros, I. Tsang, S. Gjessing, G. Fairhurst, C. Griwodz, and M. Welzl, "Reducing internet latency: A survey of techniques and their merits," *in IEEE Communications Surveys Tutorials*, vol. 18, 2016.
[2] T. Hoff, "Latency is everywhere and it costs you sales-how to crush it," *in High Scalability, July*, vol. 25, 2009.
[3] E. Schurman and J. Brutlag, "Performance related changes and their user impact," in *velocity web performance and operations conference*, 2009.
[4] B. Ager, W. Mühlbauer, G. Smaragdakis, and S. Uhlig, "Comparing dns resolvers in the wild," in *Proc.of IMC*, 2010.
[5] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, "Demystifying page load performance with wprof," in *Proc. of NSDI*, 2013.
[6] A. Flavel, P. Mani, D. Maltz, N. Holt, J. Liu, Y. Chen, and O. Surmachev, "Fastroute: A scalable load-aware anycast routing architecture for modern cdns," in *Proc. of NSDI*, 2015.
[7] M. Prince. (2011, Oct) A brief primer on anycast. [Online]. Available: https://blog.cloudflare.com/a-brief-anycast-primer/
[8] F. Chen, R. K. Sitaraman, and M. Torres, "End-user mapping: Next generation request routing for content delivery," in *Proc. of SIGCOMM*, 2015.
[9] Y. Rekhter, T. Li, and S. Hares, "A border gateway protocol 4 (bgp-4)," Tech. Rep., 2005.
[10] Z. Li, D. Levin, N. Spring, and B. Bhattacharjee, "Internet anycast: Performance, problems, and potential," in *Proc. of SIGCOMM*, 2018.
[11] M. Lentz, D. Levin, J. Castonguay, N. Spring, and B. Bhattacharjee, "D-mystifying the d-root address change," in *Proc. of IMC*, 2013.
[12] M. Calder, A. Flavel, E. Katz-Bassett, R. Mahajan, and J. Padhye, "Analyzing the performance of an anycast cdn," in *Proc. of IMC*, 2015.
[13] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein, "Maglev: A fast and reliable software network load balancer," in *Proc. of NSDI*, 2016.
[14] V. K. Adhikari, Y. Guo, F. Hao, V. Hilt, and Z. Zhang, "A tale of three cdns: An active measurement study of hulu and its cdns," in *Proc. of INFOCOM*, 2012.
[15] D. Eastlake and T. Hansen, "US Secure Hash Algorithms (SHA and HMAC-SHA)." [Online]. Available: http://www.ietf.org/rfc/rfc4634.txt
[16] E. Nygren, R. K. Sitaraman, and J. Sun, "The akamai network: a platform for high-performance internet applications," in *Proc. of SIGOPS*, 2010.
[17] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, Aug. 2018. [Online]. Available: https://rfc-editor.org/rfc/rfc8446.txt
[18] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," Tech. Rep. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-17
[19] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of open vswitch," in *Proc. of NSDI*, 2015.
[20] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *Proc. of ATC*, 2010.
[21] J. Jung, E. Sit, H. Balakrishnan, and R. Morris, "Dns performance and the effectiveness of caching," *in IEEE/ACM Transactions on Networking*, 2002.
[22] A. Singla, B. Chandrasekaran, P. B. Godfrey, and B. Maggs, "The internet at the speed of light," in *Proc. of HotNets*, 2014.
[23] M. Calder, A. Flavel, E. Katz-Bassett, R. Mahajan, and J. Padhye, "Analyzing the performance of an anycast cdn," in *Proc. of IMC*, 2015.
[24] L. Wei and J. Heidemann, "Does anycast hang up on you (udp and tcp)?" *in IEEE Transactions on Network and Service Management*, vol. 15, 2018.