A Fast Hybrid Data Sharing Framework for Hierarchical Mobile Edge Computing

Junjie Xie*, Deke Guo*‡, Xiaofeng Shi†, Haofan Cai†, Chen Qian†, Honghui Chen*
*Science and Technology on Information Systems Engineering Laboratory
National University of Defense Technology, Changsha Hunan 410073, China
†Department of Computer Science and Engineering, University of California Santa Cruz, CA 95064, USA
‡PCL Research Center of Networks and Communications, Peng Cheng Laboratory, Shenzhen, China.
Deke Guo is the corresponding author: dekeguo@nudt.edu.cn

***Abstract*—Edge computing satisfies the stringent latency requirements of data access and processing for applications running on edge devices. The data location service is a key function to provide data storage and retrieval to enable these applications. However, it still lacks research of a scalable and low-latency data location service in mobile edge computing. Meanwhile, the existing solutions, such as DNS and DHT, fail to meet the low latency requirement of mobile edge computing. This paper presents a low-latency hybrid data-sharing framework, HDS, in which the data location service is divided into two parts: intra-region and inter-region. In the intra-region part, we design a data sharing protocol called Cuckoo Summary to achieve fast data localization. In the inter-region part, we develop a geographic routing based scheme to achieve efficient data localization with only one overlay hop. The advantages of HDS include short response latency, low implementation overhead, and few false positives. We implement our HDS framework based on a P4 prototype. The experimental results show that, compared to the state-of-the-art solutions, our design achieves 50.21% shorter lookup paths and 92.75% fewer false positives.**

I. Introduction

It is predicted that the number of edge devices will grow up to 27 billion by 2021, including the Internet-of-Things (IoT) devices, mobile personal devices, and wireless sensors [1]. The prevailing cloud computing paradigm can hardly help the applications with stringent latency requirements that run on edge devices, such as virtual reality and emergency reactions. *Mobile Edge Computing* (MEC) is an emerging alternative that provides computation and storage resources over a huge number of geographically distributed edge servers close to the edge devices. The *hierarchical edge computing architecture* has been proposed to provide data and computation support for edge applications [1][2]. The hierarchy of edge servers includes a traditional wide-area cloud Data Center (DC) at its root and a large number of edge servers deployed at the end of the network. Furthermore, those edge servers are divided into different regions where each region has a small DC to manage those edge servers in the region.

Data sharing among edge devices is a unique feature of MEC, which can efficiently reduce the response latencies of data requests in MEC [3]. Data sharing in MEC can mainly be classified into two types. In the first type, the shared data is collected by edge devices [4]. The data produced by an edge device could be used by another edge device to perform some collaborative tasks. Meanwhile, the data may also be used by the data collectors themselves. The edge devices may move from their original area to another area, which further complicates data sharing in MEC. In the second type, the shared data is generated at the remote Cloud. Then, the edge servers collaboratively deliver the data to edge devices. In MEC, to achieve efficient data sharing, the *data location service* is the key function and further provides data support for many emerging applications. The *data location service* is the process of finding a server that stores a specific data item, which is used in both data storage and retrieval.

Domain Name Service (DNS) [5] and Distributed Hash Table (DHT) [6] based methods could be the potential solutions to achieve a data location service. However, they incur long latencies to respond to requests of data lookup in MEC. For example, under the DNS-based scheme, some data requests could be forwarded to the root DNS server. In this case, those lookup requests will go through long paths, which further incur long latencies. Another alternative method is a DHT-based scheme, which has been extensively studied in Peer to Peer (P2P) networks. However, the lookup requests will go through up to $log(n)$ overlay hops to locate the data. Worsely, the physical path traversed by a lookup request in MEC could be longer than the direct route to access the remote Cloud. Recent work suggests a flat architecture of MEC to reduce routing latency [7]. However, it faces a severe scalability problem and fails to achieve efficient data sharing across geographically distributed edge networks. Therefore, there is an urgent need to investigate the data location service problem under the hierarchical MEC architecture over a large number of distributed edge servers.

In this paper, we leverage the features of the hierarchical MEC architecture and propose a novel hybrid data-sharing framework, called HDS, to achieve a low-latency and scalable location service. Under the HDS framework, for a shared data item, the related server will first publish its data index to a unique region DC. All region DCs work together to maintain the global indices of all shared data items. Furthermore, the HDS partitions the data sharing into two parts: intra-region and inter-region. In detail, a data request is first processed in the local edge server. If the data can not be founded, it is then forwarded to the corresponding region DC, which will conduct

the intra-region data lookup. If the data have not been cached in the region, the data request will be further processed by the inter-region data location service. Under the HDS framework, the data from all edge servers, region DCs, and the cloud DC can all be quickly located.

The challenges for intra-region data-sharing come from three parts, namely, network bandwidth saving, fast data location service, and low memory consumption. The key is to design an efficient sharing protocol, which can support fast data location service and has low memory consumption in each region DC. The well-known protocol to achieve this is summary cache [8], which uses Bloom filter to support data lookups and sharing. However, the summary cache is inefficient due to a large number of memory accesses as well as the high false-positive rate. A false positive means the summary cache answer "yes" for a data item that has not been cached in the region, which causes a wasted lookup message and the processing cost. To overcome the drawbacks of summary cache, we design *Cuckoo Summary* to achieve efficient data sharing in intra-region. The core component of *Cuckoo Summary* is a Cuckoo hash table [9] where each entry includes the fingerprint of a cached data item and the identifier of the edge server that stores the data. The *Cuckoo Summary* can achieve not only higher lookup throughput but also fewer false positives.

The challenges for inter-region data sharing include short lookup path and low implementation overhead. To achieve the data location service across regions, we design a geographic routing based scheme to obtain the location of a requested data in inter-region. This method utilizes the advantages of Multi-hop Delaunay Triangulation (MDT) [10] and software-defined networking (SDN) [11][12], and it is called MDT-based scheme. More precisely, a virtual space is first maintained in the control plane. Then switches that are directly connected to region DCs will be assigned coordinates in the virtual space. After that, data requests will be forwarded based on their positions in the virtual space. A lookup request can be directly delivered from the ingress region DC to the destination region DC with only one overlay hop. Meanwhile, the MDT-based scheme has low implementation overhead. That is, only a few forwarding entries are needed in each switch to support the MDT-based scheme.

We implement our HDS framework, which consists of the *Cuckoo Summary* in intra-region and the MDT-based scheme in inter-region, on a P4 prototype. The experiment results demonstrate the advantages of HDS framework. More precisely, our design achieves 50.21% shorter lookup paths and 92.75% fewer false positives than the state-of-the-art solutions.

The rest of this paper is organized as follows. Section II introduces the related work of this paper. In section III, we present the system overview. Section IV details the design of intra-region data sharing. We present the design of inter-region data location service in Section V. Section VI shows the performance evaluation including a prototype implementation and large-scale simulations. We conclude this paper in Section VII.

II. Related work

In the MEC environment, a lot of edge servers are deployed at the network edge to provide the computing and storage capacity for edge devices. In this case, data sharing is essential to reduce the latency of data retrieval, and the data location service is a crucial function.

Data location. A well-known solution to the data location service is the distributed hash table (DHT), which have been extensively studied in P2P networks [13]. However, as pointed by Cox et al. [14], those DHT systems suffer from high latencies. It is because that a lookup request needs to go through $O(logN)$ overlay hops to retrieve data, and N is the number of edge servers. Recently, to reduce the latency of data retrieval, some researchers propose the hierarchical DHT architecture for information-centric networks (ICN) [15][16]. However, those hierarchical DHT architectures still employ the typical DHT scheme [6] for inter-region routing. They still need $O(logN)$ overlay hops to retrieve data across regions and further incurs long latencies. GHT incurs $O(\sqrt{N})$ routing cost for data-centric storage [17][18]. Some work can achieve $O(1)$ DHT, such as Structured Superpeers [19] and Beehive [20]. However, they need to store a large amount of index information or add many data duplicates in edge servers. They are challenging to be deployed in practice. To achieve efficient data location service, recent work suggests a flat data-sharing mechanism for edge computing [7]. However, it faces a severe challenge of scalability and fails to achieve efficient data sharing across wide area networks. Therefore, in this paper, we propose an MDT-based scheme with a low implementation overhead to achieve $O(1)$ DHT for data sharing across regions.

Cache sharing. To achieve cache sharing among Web proxies, Fan et al. propose a cache sharing protocol called summary cache [8]. Assume that s cache proxy servers want to share their cached data. Each proxy server needs to maintain s Bloom filters and checks all these Bloom filters when it receives a data request. Summary cache incurs low lookup throughput and high false-positive rate. That is, for some data that are not cached, the summary cache could still answer "yes" due to false positives. Some variants of Bloom filter could support this operation, such as coded Bloom filters [21], Combinatorial Bloom filters [22], and shifting Bloom filters [23]. However, they suffer from massive memory consumption, high false-positive rate, and low lookup throughput.

III. System overview

In this paper, we design the HDS framework by leveraging the hierarchical infrastructure of MEC [1][24][25], as shown in Figure 1. A large number of edge servers are divided into different regions. Each region has one region DC, which has more computing and storage capacity than an edge server. The main objectives of the HDS framework include fast data location service, few false positives, and low implementation overhead. Low-latency data location service is essential to meet the low latency requirement of MEC. To achieve this goal, we conduct our design considering two main requirements in MEC. First, the cached data in the same region

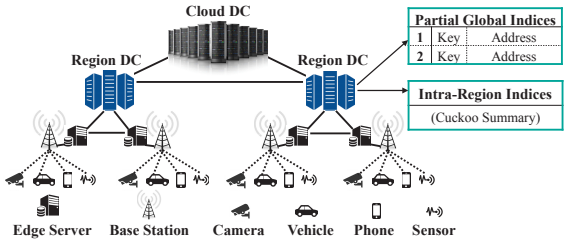


Fig. 1. The data sharing framework under the hierarchical MEC topology.

should be locally shared because it is faster to retrieve data from a neighboring edge server than from the remote Cloud. In particular, an edge server can quickly know if its neighbors have cached the requested data. Second, the data from different regions should also be efficiently shared. Furthermore, a data request should be directly forwarded from the ingress region DC to the destination region DC. Here no global information is maintained in the ingress region DC.

In this paper, we provide a general data service and do not add any more constraint on the data name or the data type. A data item can be identified by a URL, or an IoT device can identify the collected data by fitting the device ID, the date and the data type together. Under the HDS framework, the servers that have stored some shared data will publish those data indices to related region DCs. Note that those servers could be deployed inside the cloud DC or at any one edge server. The data index consists of the identifier of data and its address, which can be the IP address of the related server. More precisely, those indices of shared data are distributed among all region DCs. All region DCs collaboratively maintain the global indices of all shared data items.

As shown in Figure 1, each region DC maintains not only intra-region indices but also partial global indices. When a data request from an edge device comes, it is first forwarded to the nearest edge server through a base station (BS) or an access point (AP). If the edge server has cached the data, it immediately returns the data to the related edge device. When the requested data has not been cached in the nearest edge server, the data request will be forwarded to the corresponding region DC. The related region DC will check if itself or other edge servers in the same region has cached the requested data. If the data is still not cached in this region, it is necessary to lookup the global indices, which are distributed among all region DCs.

To achieve the intra-region data sharing, we design a sharing protocol called *Cuckoo summary*. Each edge server will send the information of all of its cached data to the corresponding region DC instead of all other edge servers in the same region. That will efficiently reduce the bandwidth consumption inside each region. Furthermore, to reduce the memory consumption, the region DC only keeps a summary of all cached data in the region. By checking the summary, the region DC can immediately know if the data is cached in this region and which edge server has cached the data. This operation is called the multi-set membership filter and lookup. The core component of *Cuckoo Summary* is a Cuckoo hash table [26], which is maintained in related region DCs. Each entry in the Cuckoo hash table is composed of the fingerprint of a cached data item and the identifier of the related edge server. The *Cuckoo Summary* is efficient due to not only less memory usage but also less number of memory accesses, compared with the well-known protocol of summary cache [8].

Another advantage of HDS is that a lookup request only goes through one overlay hop from the ingress region DC to the destination region DC for the inter-region data location service. Meanwhile, it keeps a low implementation overload in the related switches and region DCs. To achieve the fast inter-region data location service, we design an MDT-based scheme, which mainly consists of the control plane and the switch plane. The control plane maintains a virtual 2-dimensional (2D) space where each switch directly connected to a region DC will be assigned a coordinate in the virtual space. Furthermore, the control plane constructs a Delaunay Triangulation (DT) graph [10] to connect those coordinates of switches. Meanwhile, those shared data items will also be mapped into the virtual space. Then, those data requests will be forwarded based on the DT graph in the data plane. Note that the MDT-based scheme is used for achieving the inter-region data location service as well as publishing data indices. That is, when publishing a data index, the data index will be forwarded to the switch that is closest to the coordinate of the data index in the virtual space. Last, the switch will send the data index to its region DC to respond to all lookup requests of the data item.

IV. The Intra-Region Data Sharing

The data requests follow some locality patterns due to underlying usage patterns and human interest [15]. The observation further motivates the data sharing inside each region. First, users staying in the same area (e.g., same company, same institution, same country) exhibit similarities in their requested data. This effect can be exploited by caching requested data and serving subsequent requests with the cached copy in the region. Second, users typically have a relatively high interest in local data from their local organization, which can respond to requests with their own organization's domain. That is called the neighborhood effect. Furthermore, supporting data locality has significant potential to reduce costly inter-region traffic, overall network traffic, and latency.

The objective is to achieve data sharing among edge servers inside each region. A straightforward way is that each edge server can transfer the directory of cached data to all other edge servers. However, this method will waste too much network bandwidth. Therefore, we prefer to construct a centralized indexing scheme for each region and further design an intra-region data sharing protocol of *Cuckoo summary*. Each edge server sends its indexing information to the corresponding region DC, which is a small DC and has more capacity than an edge server. Furthermore, the region DC maintains a Cuckoo hash table [27], which consists of an array of buckets, and each bucket can store multiple entries. For example, Fig.

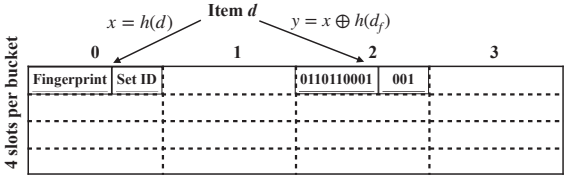


Fig. 2. (2, 4)-hash table in Cuckoo Summary where each entry consists of a fingerprint (10 bits) and a set ID (3 bits).

2 shows a (2, 4)-Cuckoo hash table, where each entry has 2 candidate buckets, x and y, and each bucket has 4 slots. Therefore, an entry can be stored in one free slot of 8 slots. To reduce memory consumption, we utilize the design of partial-key Cuckoo hashing [26][27][28]. Specifically, we store the fingerprint of a cached data in the Cuckoo hash table with only a few bits instead of storing its full identifier.

Furthermore, to fast know which edge server caches the requested data, we concatenate the set ID with the fingerprint of a cached data item. As shown in Fig. 2, the basic unit stored in the Cuckoo hash table is an entry, which consists of the fingerprint of a cached data item and its set ID. The *set ID* is the serial number of the edge server storing the data in the region. The size of the set ID is $log_2(s)$ bits where s is the maximum number of edge servers in one region. *Next, we use the set ID to denote the identifier of the corresponding edge server in intra-region*. Although the set ID could add a little memory consumption, it will efficiently improve the lookup throughput. Meanwhile, our experiment results show that Cuckoo summary achieves higher lookup throughput and fewer false positives than the state-of-the-art solutions in Section VI. Note that the evaluation is conducted when they get the same memory allocation. Under the Cuckoo summary, when a region DC receives a data request, it will first lookup the summary to check if the data has been cached in the region. If yes, it needs to answer which edge servers have this data. We treat each edge server as a set and the cached data in the edge server as the elements in the set. The procedure of data checking is called a multi-set membership filter and lookup. Next, we will describe how Cuckoo Summary performs Insert, Lookup, and Delete operations for those cached data items.

A. Cache data items

In our design, we assume that each data item has a unique data identifier. When an edge server caches a data item, it will send the insertion information to the corresponding region node. The region node first gets the fingerprint of this item by hashing its identifier. Note that the fingerprint is significantly shorter than the identifier of a data item and contains only a few bits, which can efficiently reduce the memory requirement. Then, an entry is built by concatenating the fingerprint with its set ID, which indicates which edge server caches the data. According to Cuckoo hashing [9], each item has two candidate buckets x and y that can be calculated based on Equation (1). If we use more hash functions, there will be more memory accesses, which will increase the lookup latency. Fig. 2 shows the example of inserting a new item d into a hash table of 4 buckets where each bucket has 4 slots and can store 4 entries. In Fig. 2, item d can be placed in either bucket 0 or bucket 2. If one of d's two buckets has an empty slot, Cuckoo Summary inserts d to that free slot and completes the insertion process. If neither bucket has a free slot, the Cuckoo Summary randomly selects one of the candidate buckets, kicks out an existing item and re-inserts it to another alternate location. This procedure may repeat until an empty bucket is found, or until a maximum number of displacements is reached (e.g., 300 times). If no empty bucket is found, this hash table is considered too full to insert. In this case, to accommodate more data items, it is necessary to increase the number of buckets. Although cuckoo hashing may execute a sequence of displacements, its amortized insertion time is $O(1)$ [27].

$$\begin{aligned} x &= h(d), \\ y &= x \oplus h(d_f). \end{aligned} \tag{1}$$

The xor operation in Equation (1) ensures an important property: for any data item d, its alternate bucket y can be directly calculated from the current bucket index x and the fingerprint d_f stored in bucket x. Meanwhile, x can also be calculated as follows.

$$x = y \oplus h(d_f) \tag{2}$$

Therefore, a re-insertion operation only uses information in the hash table and never has to retrieve the original item d.

B. Respond to data requests

Under the Cuckoo summary, when a region DC receives a data request, it will lookup its Cuckoo hash table to answer if the data is cached in the region. For any data item, the lookup process against a cuckoo summary is simple. Recall that each entry in the Cuckoo hash table consists of the fingerprint of a data item and its set ID, which indicates the edge server storing the data. Given an item d, the corresponding region DC first calculates d's fingerprint and two candidate buckets according to Equation (1). Then these two buckets are checked. If any existing fingerprint in either bucket matches d's fingerprint, the cuckoo summary returns the corresponding set ID. Otherwise, the summary returns false. Notice that this ensures no false-negative responses as long as bucket overflow never occurs [27]. Besides, the same fingerprint may be matched with multiple set IDs, which means multiple edge servers could have the requested data. This scenario can also occur in "summary cache" [8]. After that, the data request will be forwarded to those matched edge servers at the same time. If there are multiple data copies in this region, the user will receive the data from the edge server that responds to the request fastest.

C. Remove data items

Consider that the edge server has limited capacity. The region DC needs to delete the corresponding entry from the Cuckoo hash table when an edge server in the related region removes a cached data item. The deletion process under the

Cuckoo summary is as follows. When an edge server removes a cached data item, it will send a deletion message to the related region DC. The region DC first builds a queried entry, which consists of the fingerprint of the deleted data and its set ID. Then, it checks both candidate buckets for the queried entry; if any bucket matches, one copy of that matched entry is removed from that bucket. The deletion operation completes. Note that the deleted item must have been previously inserted. This requirement also holds for all other deletion-supporting data structures [8][23][27]. Otherwise, deleting a non-inserted item might unintentionally remove a real, different item from the same edge server that happens to share the same fingerprint. In addition, other data structures with similar deletion processes exhibit higher complex than Cuckoo summary. For example, shifting Bloom filters [23] and Summary cache [8] must use extra counters to prevent the "false deletion" problem caused by hash collisions. Those counters will further incur extra memory consumption.

D. Analysis of Cuckoo Summary

Note that we utilize the partial-key cuckoo hashing [27] to store the related data items into the Cuckoo hash table, which consists of a given number of buckets. To save memory consumption, we just store the fingerprint of a cached data item instead of storing the full key in the Cuckoo hash table. In particular, two different items d_1 and d_2 could have the same fingerprint. The Cuckoo Summary can accommodate the same fingerprint appearing multiple times in a bucket. However, like cuckoo filter [27], Cuckoo Summary is not suitable for applications that insert the same fingerprint more than $2b$ times (b is the bucket size). Otherwise, the two buckets for this duplicated item will become overloaded. There are several solutions for such a scenario. On the one hand, we can increase the bucket size. On the other hand, we can also increase the length of the fingerprint to reduce the probability of hash collision.

Furthermore, when we lookup a data item that has not been cached in a region, a false positive occurs if this data has the same fingerprint with a cached data item. Note that the fingerprint size depends only on the desired false positive probability. Like other filters, there is no false negative in our Cuckoo Summary when there is no overloaded bucket.

The probability of false positive. With larger buckets, each lookup checks more entries and thus has more chance to meet fingerprint collisions. In the worst case, a query must probe two buckets, each of which has b entries. For each entry, the probability that a query is matched against the one stored fingerprint and returns a false-positive successful match is at most $1/2^f$ where f is the fingerprint size. After making $2b$ such comparisons, the upper bound of the total probability of a false fingerprint hit is

$$1-\left(1-1/2^f\right)^{2b} \approx 2b/2^f; \tag{3}$$

which is proportional to the bucket size b. Given a target false-positive rate ϵ, the Cuckoo Summary needs to ensure $2b/2^f \leq \epsilon$. Thus the minimal fingerprint size required is approximately:

$$f \geq \lceil log_2(2b/\epsilon) \rceil = \lceil log_2(1/\epsilon) + log_2(2b) \rceil \quad bits. \tag{4}$$

We can determine the fingerprint size based on Equation (4). For example, if we desire the false positive rate is lower than 1%, $f \geq \lceil log_2(2b/0.01) \rceil = \lceil 9.64 \rceil = 10$ bits, where each bucket has $b=4$ slots, which is a recommended setting for many Cuckoo hashing based designs [26][27][29][30] and achieves a good trade-off between the space efficiency and the false positive rate.

V. The inter-region data sharing

When a requested data can not be found in a local region, it is necessary to lookup the data across other regions. In MEC, a large amount of data is stored in those geographically distributed edge servers and the remote Cloud. To efficiently lookup those data, we will construct a distributed indexing mechanism to publish those indices of shared data items to the related region DCs instead of storing all data indices in the remote Cloud. After that, those data indices are distributed among those region DCs. When we need to retrieve a data item across other regions, we first find the region DC, which stores the corresponding data index. Then, we can achieve the data location from its index. To retrieve the index of requested data, prior hierarchical architectures [15][16] adopt the typical DHT methods to realize the data lookup across regions. However, those methods incur the long response latencies because each lookup involves $log(n)$ overlay hops. To achieve fast data location service across regions, in this paper, we design an MDT-based scheme by leveraging the advantages of MDT [10] and SDN [11][31] to achieve $O(1)$ DHT with low implementation overload.

In SDN, the network consists of the control plane and the data plane, which has been successfully deployed for the inter-region or inter-data center communication [32][33]. Under the HDS framework, we do not restrict the network type in the intra-region, which can be a traditional IP based network or the software-defined networking. We just deploy SDN for the inter-region communication. Consider that the number of region DCs is not too many, our HDS framework has a good scalability. In this case, the main functions of our MDT-based scheme run in the control plane and the data plane as follows.

A. Publish data indices

The control plane maintains a virtual 2D space Ω. The indices of all shared data items and those switches directly connected to region DCs are assigned coordinates in the virtual space. Note that the coordinate of a data index can be achieved by hashing its identifier. Specifically, we use the hash function *SHA-256*, whose output is a 32-byte binary value. We only use the last 8 bytes of the hash value and convert them to two 4-byte binary numbers as the coordinate of a data index in the 2D space. Furthermore, a data index will be stored in the region DC, which is directly connected to the switch closest to the data index in the 2D virtual space. Assume that z related switches are directly connected to region DCs and

have their coordinates $\{r_i\}_{i=1}^{z}$ in the virtual space. For a data index, it is mapped to point p in the virtual space. Then, the data index will be stored in the region DC directly connected to switch r_j, where $\{|p-r_j|<|p-r_i|, i=1\ldots z, i\neq j\}$. Accordingly, those switches $\{r_i\}_{i=1}^{z}$ partition the 2D space into z convex polygons, which is called the Voronoi Tessellation [34].

B. Forward data indices

Those switches conduct greedy forwardings to keep a low implementation overhead. That is, each switch forwards a data index only based on the coordinates of the data index and its neighboring switches. No more information is needed. More precisely, a switch will forward a data index to its neighbor, which is closest to the data index in the virtual space. However, the greedy routing could be trapped in a local minimum. To provide guaranteed delivery, we utilize the property of MDT [10]. That is, given a position p in the 2D coordination, the greedy forwarding always succeeds to find a point nearest to p. To achieve this goal, the control plane constructs a Delaunay Triangulation (DT) graph to connect those coordinates of switches in the virtual space. Then, based on those connections in the DT graph, the control plane inserts forwarding entries into those switches where each forwarding entry indicates the coordinate of a neighboring switch. After that, the data request can be directly delivered from an ingress region DC to its destination region DC, and no other region DCs are involved. Therefore, the MDT-based scheme can achieve a $O(1)$ DHT.

C. Optimize switches' coordinates

Furthermore, to ensure that the path selected by the greedy routing is equal to or close to the shortest path between the ingress region DC and the destination region DC, we utilize the theory of multidimensional scaling [35] to embed the network paths among switches into the distances between coordinates in the virtual space. Specifically, the coordinate matrix Q of switches can be derived by the following equation.

$$QQ' = -\frac{1}{2}JB^{(2)}J \tag{5}$$

In Equation (5), B is the shortest path matrix among those switches. The matrix $J{=}I{-}\frac{1}{z}A$, where A is the squared matrix with all elements are 1. Like other SDN applications [32][33], the MDT-based scheme employs one or multiple controllers in the cloud DC or a region DC to collect the network state and link information among region DCs. The information is used to calculate the shortest path matrix B among those switches. Then, the coordinate matrix Q can be calculated by the eigenvalue decomposition from matrix B. By embedding the network paths, the switches' coordinates can be determined, and further, the distances between those coordinates are proportional to the physical path lengths among the corresponding switches.

D. Lookup data indices

When publishing a data index, the data index is first forwarded to the switch that is nearest to the data index in the virtual space. Then, the switch forwards the data index to its region DC, which is directly connected to the switch. After that, the region DC stores the data index and responds to all requests of the data. The lookup procedure is similar to the publishing of a data index. As shown in Fig. 3, the coordinate of data d_1 is closest to the coordinate of switch r_7. Therefore, the index of data d_1 is stored in the region DC, which is directly connected to switch r_7. When the region DC connected to switch r_1 needs to lookup the index of data d_1, the lookup request is first forwarded to switch r_1. Switch r_1 compares the distances from its neighbors to the position of d_1 in the virtual space and forwards the request to switch r_5 because switch r_5 is nearest to data d_1. Then, switch r_5 greedily forwards the request to switch r_7, which is closest to the coordinate of d_1 in the whole virtual space. Therefore, switch r_7 forwards the data request to its region DC. The region DC connected to switch r_1 can achieve the location of d_1 based on its index. Then it can retrieve the data by the shortest path routing or other more efficient routing schemes, which is orthogonal with this paper.

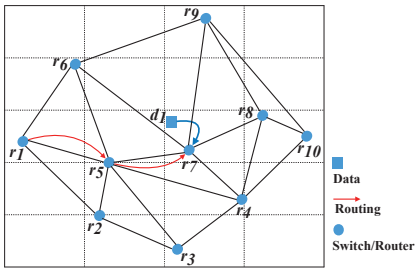


Fig. 3. The greedy routing based on MDT in inter-region.

Besides, under our MDT-based scheme, it is easy to support multiple index copies, which is helpful to the fault tolerance and the load balance of the system. For example, the system needs to maintain multiple index copies for each shared data item. We just need to concatenate a serial number with the data index for each index copy. Then, by hashing the updated index identifier, we can get its position in the virtual space. Based on their positions, those index copies will be stored in different region DCs to respond to users' requests.

VI. Performance Evaluation

We evaluate the performance of our HDS framework by a prototype implementation and large-scale simulations.

A. Implementation and prototype evaluation

We implement our HDS framework consisting of the *Cuckoo Summary* and the MDT-based scheme (CS+MDT) on a small-scale testbed. The network consists of 40 edge servers, which are divided into 10 regions, as shown in Fig. 4. In detail, 10 physical servers are employed as the region DCs. Meanwhile, 4 virtual machines (VMs) run in one physical server, and each VM is considered as an edge server. Those servers are connected by P4 switches [36][37], which are used to support the functions of SDN. We compare our HDS framework with the state-of-the-art solution, which consists

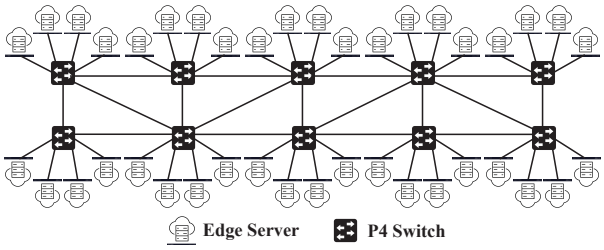
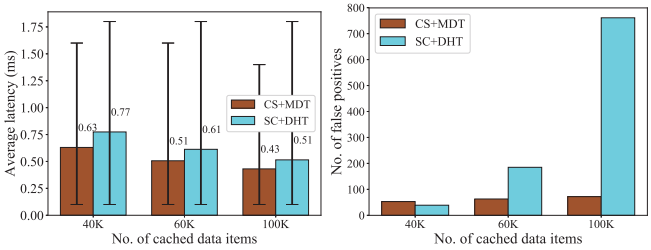


Fig. 4. The network topology consists of 40 edge servers and 10 P4 switches on a small-scale testbed.



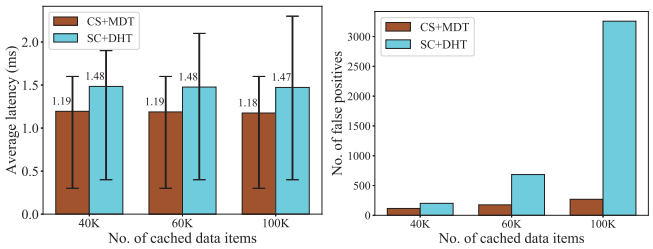
(a) The average latency of data requests. (b) The number of false positives.

Fig. 5. The data requests follow a Zipf distribution.

of the summary cache [8] and the DHT-based scheme [16] (SC+DHT). We consider that 1 million data items are first stored in the edge network. Note that the main objective of the HDS framework is to fast locate a data item no matter how the data is stored in the network. Each edge server uses the least-recently-used (LRU) as the cache replacement algorithm. Note that other cache replacement algorithms can also be employed under our HDS framework and are orthogonal with our work.

Each edge server sends $100K$ data requests, and their response latencies are recorded. Then, the average response latency is calculated. We vary the number of cached data from $40K$ to $100K$ in one region. Meanwhile, we recorded the false positives of intra-region lookups under different solutions. Note that the false positives will result in wasted lookup messages and long response latencies. We first send the data requests, which follow a Zipf distribution [38]with the exponent α=1. Note that the distribution is more intensive when α increases. In this case, more requests can be served in the intra-region. Otherwise, more requests will be forwarded into the inter-region. Fig. 5(a) shows that our HDS framework (CS+MDT) achieves shorter response latencies than the state-of-the-art solution (SC+DHT) under different numbers of cached data. It is mainly because that our MDT-based scheme achieves shorter inter-region lookup paths than the DHT-based method. Meanwhile, we can find that the number of false positives grows up as the number of cached data increases from Fig. 5(b). Note that the same memory space is allocated for our Cuckoo summary and the summary cache to conduct the intra-region data sharing.

Furthermore, we also measure the performance of HDS framework when data requests follow a uniform distribution. Fig. 6 shows the same trend as Fig. 5. More precisely, our



(a) The average latency of data requests. (b) The number of false positives.

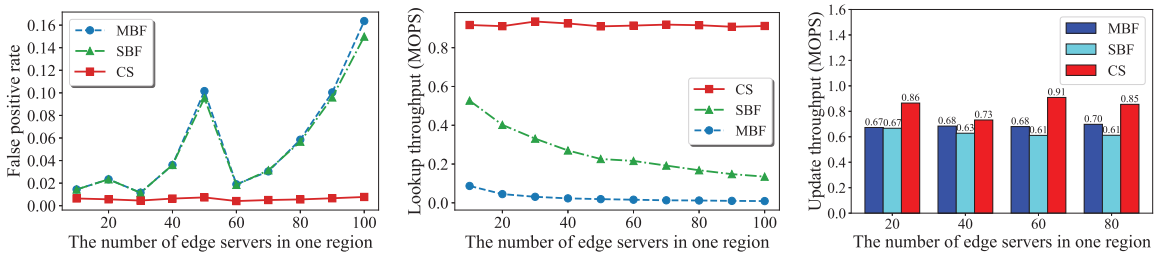
Fig. 6. The data requests follow a uniform distribution.

HDS framework (CS+MDT) always achieves shorter response latency and less number of false positives than the state-of-the-art solution (SC+DHT). Note that the differences in response latencies under different solutions are not very large. It is mainly because the scale of our testbed is small. In practice, a large number of edge servers are geographically distributed across wide area networks. In this case, the advantage of our HDS framework will be more obvious due to shorter physical paths to get data locations. In addition, compared Fig. 5(a) with Fig. 6(a), we can find that the increasing number of cached data can reduce the response latency of data requests when those requests follow a Zipf distribution. However, the reduction of response latency is not obvious when those requests follow a uniform distribution. It is because those cached data can efficiently reduce the inter-region lookups and further reduce the response latency under the Zipf distribution.

B. Large-scale simulations

1) Evaluation for intra-region data sharing: We implement *Cuckoo Summary* using Java code and compare it with alternative data structures, Multiple Bloom filters adopted in summary cache [8] and the state-of-the-art Shifting Bloom filter [23], for multi-set membership filter and lookup. The region DC keeps a compact summary of the cached data from all edge servers in the same region. When a region DC receives a data request, it first checks the summary to see if the request can be matched at any edge server in the region. A random number generator assigns each cached data a 64-bit data identifier. We did not eliminate duplicated data identifiers because the probability of collision is very small for the random integer generator. Meanwhile, multiple data copies may exist in one region in MEC to meet the stringent latency requirement. The performance metrics include the false positive rate, the lookup throughput, and the update throughput. The compared data structures are as follows.

- *Cuckoo Summary (CS)* consists of one (2, 4)-Cuckoo hash table. Each cached data item has 2 candidate buckets, and each bucket has 4 slots, each of which can accommodate an entry. That is a recommended setting in many applications of partial-key Cuckoo hashing [27][26][39][30]. Per entry in CS is the fingerprint of a data item and the set identifier that indicates the involved edge server.



(a) The false positive rate. (b) The lookup throughput. (c) The update throughput.

Fig. 7. The performance comparison of different solutions to the intra-region data sharing.

- *Multiple Bloom filter (MBF)* [8] is composed of s Bloom filters, and each Bloom filter needs to use k hash functions. Each Bloom filter is a compact summary of the cached data in one edge server, and s denotes the number of edge servers in one region.
- *Shifting Bloom filter (SBF)* [23] is composed of one Bloom filter and k hash functions, and uses offsets to record the information of the set identifier. To insert an element, it maps the element to k positions in the bit array, offsetting the k positions by a certain amount relevant to the set identifier, and then set the k new positions to 1. To query an element, it performs k hash computations and checks s bits after these k positions.

False positive rate. There is a trade-off between the false positive rate and memory consumption. For those Bloom filter-based data structures, they can achieve a lower false-positive rate when consuming more memory. Therefore, we compare different data structures when they have the same memory allocation. We test false-positive rates of different data structures when varying the number of edge servers from 10 to 100 in one region. Accordingly, the number of cached data items varies from $100K$ to $1M$. Note that more memory is allocated for each summary when more cached data needs to be inserted into those summaries.

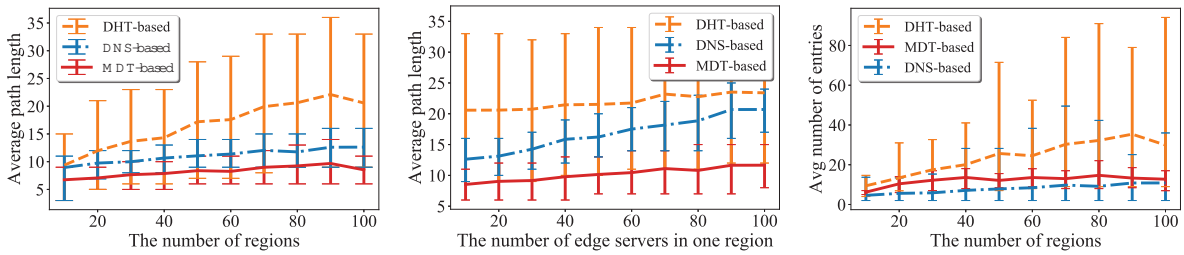
We can see that the false positive rate of our CS is almost stable when the number of edge servers varies from Fig. 7(a) and is lower than the false-positive rates of the other two data structures. It is because that the false positive of the CS is only related to the length of the fingerprint, as analyzed in Section IV-D. Note that the false-positive rates of MBF and SBF decrease from 50 edge servers to 60 edge servers in Fig. 7(a). When the number of edge servers varies from 50 to 60 in one region, those summaries need to accommodate more cached data. Therefore, we allocate more memory for those data structures to accommodate those cached data items. After that, those summaries have less level of occupancy when there are 60 edge servers than that there are 50 edge servers. More precisely, when there are 50 edge servers in the region, the occupancy rate of CS is 95.37%. *The occupancy rate is defined as the ratio of used slots to the total amount of available slots in the CS.* Accordingly, there are 4.63% free slots in the Cuckoo hash table for our CS. However, the occupancy rate is only 57.22% when 60 edge servers exist in the region. It is worth noting that when 50 edge servers exist in the edge network, our CS achieves 92.28% and 92.75% less false positives than SBF and MBF data structures, respectively.

Lookup throughput. This section compares the lookup throughput, and each point is the average of 10 runs. We first insert 1 million items into those summaries of different data structures and then conduct massive lookup operations. In Fig. 7(b), we can see that under our *CS*, the region DC can achieve obviously higher lookup throughput than other alternative data structures. Meanwhile, we can see that the lookup throughputs under the SBF and MBF decrease as the increase of the number of edge servers. It is because that the MBF needs to check more Bloom filters, and the SBF needs more memory accesses to get more bit values when there are more edge servers. However, our *CS* is independent of the number of edge servers. For any one data request, the lookup in intra-region, the *CS* only needs 2 memory accesses to check if the data is stored in this region.

Update throughput. The data replacement is very common in MEC due to the limited capacity of the edge server and the age of information [40]. In this case, the region DC needs to delete prior cached data item and insert a new data item into the Cuckoo summary. Therefore, the update throughput is also crucial to the intra-region data sharing. Fig. 7(c) shows that our *CS* can achieve higher update throughput than MBF and SBF structures. The update throughputs of the three data structures have nothing to do with the number of edge servers in Fig. 7(c). Meanwhile, we note that the update throughput of the *CS* structure decreases when the number of edge servers varies from 60 to 80. It is because that the Cuckoo hash table has higher occupancy rate in the *CS*. In detail, the occupancy rates are 57.22% and 76.29% when 60 and 80 edge servers exist in the region, respectively. Therefore, to achieve a high update throughput, we recommend that the occupancy rate of the *CS* is under 90%.

2) Evaluation for hybrid data sharing: In this section, we evaluate the performances of hybrid data sharing frameworks. The performance metrics include the path lengths of lookup requests and the number of forwarding entries in switches. If a requested data can not be found in intra-region, its lookup path consists of intra-region and inter-region paths. In intra-region, the shortest path routing is employed. For data location services across regions, the compared schemes are as follows.

- The MDT-based scheme is designed in Section V. Each switch conducts greedy forwarding based on the coordi-



(a) The path lengths of different schemes where each region includes 10 edge servers.

(b) The path lengths of different schemes where the network consists of 100 regions.

(c) The number of forwarding entries for the data location service across regions.

Fig. 8. The performance comparision of hybrid data sharing frameworks under different network sizes.

nates of its neighbors and the requested data.

- The DHT-based scheme is implemented based on a distributed hash table. Each region DC maintains its fingertable [6][16] to realize the data location service.
- The DNS-based scheme is a hierarchical indexing mechanism. If a requested data can not be found in a local region, the request will be forwarded to the Cloud DC.

The path lengths of lookup requests. We evaluate the path lengths of lookup requests under different data sharing schemes where the paths in the hybrid system include the paths in intra-region and the paths across regions. The network consists of one cloud DC, region DCs, and edge servers. Each edge server is connected to the nearest region DC. Then, each edge server lookups the data from all other edge servers. All lookup paths are recorded, and then the average length of lookup paths is calculated under each network setting. We first evaluate the impact of the number of regions on the path lengths of lookup requests. The number of regions varies from 10 to 100 where each region include 10 edge servers. Fig. 8(a) shows that our *MDT-based* scheme achieves significantly shorter paths than DNS-based and DHT-based schemes. Consider those edge servers are geographically distributed across wide area networks, and shorter path lengths mean faster responses to those data requests. Under the DHT-based scheme, the increase of the number of regions results in the obvious increase of the path lengths. It is mainly because the path length for inter-region lookup increases under the DHT-based scheme.

Furthermore, we evaluate the impact of the number of edge servers in one region on the path lengths of lookup requests. The network consists of 100 regions, and the number of edge servers varies from 10 to 100 in one region. That is, the total number of edge servers varies from $1,000$ to $10,000$ in the whole network. Fig. 8(b) shows that the path lengths under the DNS-based scheme increase when the number of edge servers increases in one region. It is because that the network size is large and some lookup requests could be forwarded to the remote Cloud that results in the long lookup paths. However, the number of edge servers in one region has a little influence on the path lengths of MDT-based and DHT-based schemes. More precisely, when the network consists of $10,000$ edge servers that are divided into 100 regions, our *MDT-based* scheme achieves 43.70% and 50.21% shorter path lengths than DNS-based and DHT-based schemes, respectively.

The number of forwarding entries. Furthermore, we evaluate the number of forwarding entries in switches for supporting the data location service across regions where less number of forwarding entries mean less implementation overhead [41][42]. Although the DNS-based scheme uses less forwarding entries, it is a centralized scheme. Worsely, there is significant load imbalance among switches under the DHT-based and DNS-based scheme. That is, the number of forwarding entries in some switches is significantly more than other switches. Fig. 8(c) shows that our *MDT-based* scheme costs less forwarding entries to support the data lookup across regions than the DHT-based scheme. Meanwhile, the numbers of forwarding entries under DNS-based and DHT-based schemes grow up as the increasing number of edge servers in Fig.8(c). In addition, the number of edge servers has a modest impact on the number of forwarding entries under the MDT-based scheme, and the number of forwarding entries in a switch is only related to the number of neighbors of the switch.

VII. Conclusion

In this paper, we design the HDS framework for the hierarchical MEC. The HDS consists of an intra-region data sharing protocol called *Cuckoo Summary* and the MDT-based scheme for the data location service across regions. The advantages of HDS framework include short lookup path, few false positive, and high lookup throughput. The prototype implementation and large-scale simulations show that our design achieves 50.21% shorter lookup paths and 92.75% fewer false positives than the state-of-the-art solutions.

Acknowledgment

We would like to thank the anonymous reviewers for their detailed comments and suggestions. This work is partially supported by National Natural Science Foundation of China under Grant Nos. U19B2024 and 61772544, National key research and development program under Grant Nos. 2018YFB1800203 and 2018YFE0207600, the project FANet under Grant Nos. PCL2018KP001 and LZC0019.

REFERENCES

[1] A. Tiwari, B. Ramprasad, S. H. Mortazavi, M. Gabel, and E. d. Lara, "Reconfigurable streaming for the mobile edge," in *Proc. of ACM HotMobile*, 2019, pp. 153–158.

[2] S. H. Mortazavi, M. Salehe, C. S. Gomes, C. Phillips, and E. de Lara, "Cloudpath: A multi-tier cloud computing framework," in *Proc. the Second ACM/IEEE SEC*, 2017, pp. 20:1–20:13.

[3] E. Bastug, M. Bennis, and M. Debbah, "Living on the edge: The role of proactive caching in 5g wireless networks," *IEEE Communications Magazine*, vol. 52, no. 8, pp. 82–89, 2014.

[4] M. Wang, C. Qian, X. Li, and S. Shi, "Collaborative validation of public-key certificates for iot by distributed caching," in *Proc. of IEEE INFOCOM*, April 2019, pp. 1–9.

[5] E. Nygren, R. K. Sitaraman, and J. Sun, "The akamai network: A platform for high-performance internet applications," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 3, pp. 2–19, Aug. 2010.

[6] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proc. of ACM SIGCOMM*, 2001, pp. 149–160.

[7] J. Xie, C. Qian, D. Guo, M. Wang, S. Shi, and H. Chen, "Efficient indexing mechanism for unstructured data sharing systems in edge computing," in *Proc. of IEEE INFOCOM*, April 2019.

[8] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," in *SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 4. ACM, 1998, pp. 254–265.

[9] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122 – 144, 2004.

[10] S. S. Lam and C. Qian, "Geographic routing in d-dimensional spaces with guaranteed delivery and low stretch," *SIGMETRICS Perform. Eval. Rev.*, vol. 39, no. 1, pp. 217–228, Jun. 2011.

[11] J. Xie, D. Guo, Z. Hu, T. Qu, and P. Lv, "Control plane of software defined networks: A survey," *ELSEVIER Computer Communications*, vol. 67, pp. 1–10, 2015.

[12] J. Xie, D. Guo, C. Qian, L. Liu, B. Ren, and H. Chen, "Validation of distributed sdn control plane under uncertain failures," *IEEE/ACM TRANSACTIONS ON NETWORKING*, vol. 27, no. 3, pp. 1234–1247, 2019.

[13] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, S. Lim *et al.*, "A survey and comparison of peer-to-peer overlay network schemes." *IEEE Communications Surveys and tutorials*, vol. 7, no. 1-4, pp. 72–93, 2005.

[14] R. Cox, A. Muthitacharoen, and R. T. Morris, "Serving dns using a peer-to-peer lookup service," in *International Workshop on Peer-To-Peer Systems*. Springer, 2002, pp. 155–165.

[15] M. D'Ambrosio, C. Dannewitz, H. Karl, and V. Vercellone, "Mdht: a hierarchical name resolution service for information-centric networks," in *Proc. the ACM SIGCOMM workshop on ICN*, 2011, pp. 7–12.

[16] R. Li, H. Harai, and H. Asaeda, "An aggregatable name-based routing for energy-efficient data sharing in big data era," *IEEE Access*, vol. 3, pp. 955–966, 2015.

[17] S. Ratnasamy, B. Karp, S. Shenker, D. Estrin, R. Govindan, L. Yin, and F. Yu, "Data-centric storage in sensornets with ght, a geographic hash table," *Mobile Networks and Applications*, vol. 8, no. 4, pp. 427–442, 2003.

[18] Thang Nam Le, Wei Yu, Xiaole Bai, and Dong Xuan, "A dynamic geographic hash table for data-centric storage in sensor networks," in *IEEE WCNC 2006*, vol. 4, April 2006, pp. 2168–2174.

[19] A. T. Mizrak, Y. Cheng, V. Kumar, and S. Savage, "Structured superpeers: Leveraging heterogeneity to provide constant-time lookup," in *Proc. the Third IEEE WIAPP*, 2003, pp. 104–111.

[20] V. Ramasubramanian and E. G. Sirer, "Beehive: O (1) lookup performance for power-law query distributions in peer-to-peer overlays." in *Nsdi*, vol. 4, 2004, pp. 8–8.

[21] F. Chang, W.-c. Feng, and K. Li, "Approximate caches for packet classification," in *IEEE INFOCOM*, vol. 4, 2004, pp. 2196–2207.

[22] F. Hao, M. Kodialam, T. V. Lakshman, and H. Song, "Fast dynamic multiple-set membership testing using combinatorial bloom filters," *IEEE/ACM Trans. Netw.*, vol. 20, no. 1, pp. 295–304, Feb. 2012.

[23] T. Yang, A. X. Liu, M. Shahzad, Y. Zhong, Q. Fu, Z. Li, G. Xie, and X. Li, "A shifting bloom filter framework for set queries," *Proc. the VLDB Endowment*, vol. 9, no. 5, pp. 408–419, 2016.

[24] Milan Patel *et al.*, "Mobile-edge computing introductory technical white paper," *Mobile-edge Computing industry initiative*, 2014.

[25] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile Edge Computing A key technology towards 5G," *European Telecommunications Standards Institute White Paper*, 2015.

[26] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "Silt: A memory-efficient, high-performance key-value store," in *Proc. the 23rd ACM SOSP*, 2011, pp. 1–13.

[27] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proc. of ACM CoNEXT*, 2014, pp. 75–88.

[28] S. Shi, C. Qian, and M. Wang, "Re-designing compact-structure based forwarding for programmable networks," in *Proc. IEEE ICNP*, 2019.

[29] M. W. and M. Zhou and S. Shi and C. Qian, "Vacuum filters: More space-efficient and faster replacement for bloom and cuckoo filters," in *Proc. of the VLDB Endowment*, 2020.

[30] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, high performance ethernet forwarding with cuckooswitch," in *Proc. of the Ninth ACM CoNEXT*, 2013, pp. 97–108.

[31] J. Xie, D. Guo, X. Li, Y. Shen, and X. Jiang, "Cutting long-tail latency of routing response in software defined networks," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 3, pp. 384–396, 2018.

[32] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," in *Proc. of the ACM SIGCOMM*, 2013, pp. 15–26.

[33] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a globally-deployed software defined wan," in *Proc. of the ACM SIGCOMM*, 2013, pp. 3–14.

[34] S. Fortune, "Voronoi diagrams and Delaunay triangulations," in *Handbook of Discrete and Computational Geometry*, 2nd ed., J. E. Goodman and J. O'Rourke, Eds. CRC Press, 2004.

[35] I. Borg and P. J. Groenen, *Modern multidimensional scaling: Theory and applications*. Springer Science & Business Media, 2005.

[36] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[37] "$P4_{16}$ language specification," [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf, accessed July 2019.

[38] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distributions: evidence and implications," in *IEEE INFOCOM*, vol. 1, March 1999, pp. 126–134.

[39] B. Fan, D. G. Andersen, and M. Kaminsky, "Memc3: Compact and concurrent memcache with dumber caching and smarter hashing," in *Proc. the 10th USENIX NSDI*, Lombard, IL, 2013, pp. 371–384.

[40] C. Li, S. Li, and Y. T. Hou, "A general model for minimizing age of information at network edge," in *IEEE INFOCOM*, April 2019, pp. 118–126.

[41] Y. Yu, D. Belazzougui, C. Qian, and Q. Zhang, "A concise forwarding information base for scalable and fast name lookups," in *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, 2017.

[42] ——, "Memory-efficient and ultra-fast network lookup and forwarding using othello hashing," *IEEE/ACM Trans. Netw.*, vol. 26, no. 3, pp. 1151–1164, Jun. 2018.